RAID 5 Block Storage Implementation

Introduction and problem statement:

This project implements the practical multi server's/client files system (the file servers running on the same computer) with networking and fault tolerance with support of Redundant Array of Independent Disks (RAID5) block storage approach. RAID 5 is a data backup technology for hard disk drives that uses both disk striping and parity. It is one of the levels of RAID: Redundant Array of Independent Disks, originally Inexpensive Disks. In practice, RAID5 usually offers a good balance between security, fault tolerance, and performance, hence highly efficient for data storage. These servers just aware about the raw block layer of the filesystem. The rest of the filesystem layers and caching are implemented in the Memory_client.py. Server just exposes Get() and Put() interfaces. This project aims to distribute and store the data across the multiple data servers in order to reduce the load on each server (servers holding the copies of data), to provide large aggregated storage capacity with the fault tolerance mechanism while operation of the system. The system is robustly designed which handles the failure scenarios like Fail-stop of a server with repair mechanism and force corruption of a block in any server to emulates the failure. This system performance is analyzed by calculating the average load on the multiple servers (number of requests handled per server) against the single server system.

Design and implementation:

The system design followed step by step milestones.

- ### Support of multiple servers:

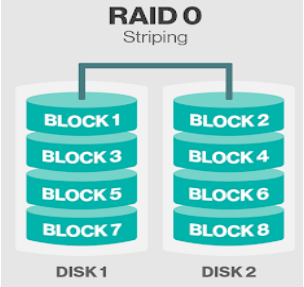
  memoryfs_server.py is modified to accept the server id ***(-sid)*** with a port ***(-port)*** to make the each server runs with unique id with unique port by adding the additional parameter to the *argparse.ArgumentParser().* In practice these parameters are also referred as IP address of the server which is listening on particular port.

  memoryfs_shell_rpc.py is modified to accept the port ***(-port0, -port1, port2, … up to n)*** numbers on which servers are up and running along with the client id ***(-cid)*** by adding the additional parameter to the *argparse.ArgumentParser().*

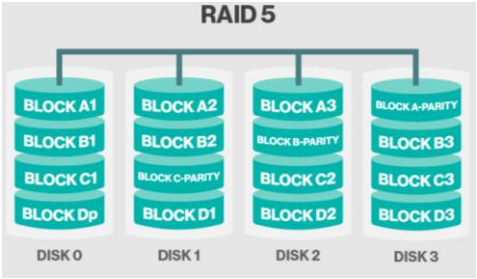
  This multiple server system is tolerated from Connection Refused Errors by adding a try/Except block where ever the requests go to the server.

- ### Virtual to Physical address conversion(RAID5 Mapping):

  The virtual address is referred to the address/block number which the client refers. The physical address is referred as the one which servers refers. Below figure shows the simple RAID0 mapping of the address.



RAID 5 incorporates striping of data just like in a RAID 0 array, however, in a RAID 5 there are redundant pieces of the data that are also distributed across the drives and are referred to as parity. Having the parity blocks staggered across each drive allows any single drive in the RAID 5 array to fail without any data loss. Each of those three data blocks have a certain number of bits that are 1's & 0's. Those values are then summed up (XORed) as a value on the parity block. Below figure shows the RAID5 disks organization with parity distribution.



Data blocks server id can be calculated using,

block_server_id = block_number % (TOTAL_NUM_SERVERS-1)

The physical block number can be calculated using (given total servers, virtual block address)

$$\textbf{ph\_block\_number} = \frac{(\boldsymbol{Block\ number})}{\boldsymbol{Total\ No\ of\ servers}-\mathbf{1}}$$

The parity server id can be calculated using (given total servers, physical block),

parity_server_id = (TOTAL_NUM_SERVERS -1) - (ph_block_number % TOTAL_NUM_SERVERS)

where,

block_number = Virtual block number referred by the client.

ph_block_number = Physical block number referred by the server.

TOTAL_NUM_SERVERS = Total number of servers running.

- ## Adding Checksum

This system implements the dedicated data structure in the *memoryfs_server.py* to store all the checksum of each raw blocks. A checksum is a small-sized block of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage. Checksums are typically used to compare two sets of data to make sure they are the same. ***Hashlib*** module in python implements a common interface to many different secure hash and message digest algorithms. ***hash_md5.hexdigest()*** will return the *hex string* representation for the digest.

Each block of the file system stores the data in the byte array format. The check sum of the data will enable to check the integrity of the data every time the client accesses this data.

The size of each hash is 128 bits (16bytes)

The size of each block is 128 bytes (so this file system)

So, each checksum block stores 128 bytes /16bytes = 8 blocks checksum.

So, for 256 blocks system, 32 checksum blocks are required.

In this implementation, the *list* named ***self.checksum_datastructure = []*** is used (for simplicity) to store the checksums of the blocks instead of creating the additional block in the system. If the number of blocks in the system is 256 then the size of the checksum list is also 256. This list will be initialized with the checksum of the 128bytes 0's in the server **DiskBlocks()** class **__init__** method while initializing the data blocks of the respective server.

The checksum initialization on the server has been done using the following lines.

```
csm = hashlib.md5(bytes(putdata)).hexdigest()

self.checksum_datastructure.append(csm)
```

The checksum of the each block will be checked on every ***GET()*** , if there is any mismatch of the checksum then the new data will be created by XORing the other servers data blocks/parity. Similarly on every **PUT()** the checksum of the data will be updated with the respective new data going into the server.

- ## Get()

The get method is exposed to the client by the server to read any block and respective checksum data. The below **pseudo code shows the Server Get** code used by the client.

```
def ServerGet(self, virtual_block):
  if virtual_block in range(0,TOTAL_NUM_BLOCKS):
                    #Calculate the physical block number , data block server id
                     and parity server id for a given virtual address
    ph_block_number,block_server_id,parity_server_id = self.MappingFunc(virtual_block)
    try:
      if block_server_id <= Total_Num_Servers:
                    #Get request to the server to retrieve the blocks data
              data = self.block_serverN.Get(ph_block_number)
      if data != -1:
        return bytearray(data)
      else:
         #On failure on getting data the data should be created using the other servers data blocks
        return self.CorrectDataBlock(block_server_id,ph_block_number)
    except ConnectionRefusedError:
         #when the server failed to respond the same data should be created using other disks.
      new_data = self.CreateFailedDataOnGet(ph_block_number,block_server_id)
      return new_data
  logging.error('ServerGet: Block number larger than TOTAL_NUM_BLOCKS: ' + str(virtual_block))
  quit()
```

The GET() method is robust to handle the failures like server failure, checksum errors etc. when any checksum mismatch happens the correct data will be calculated by retrieving the other data blocks in the same physical tripe using the function ***CorrectDataBlock(block_server_id,ph_block_number)*** . this function takes server id and physical block and returns the correct data by XORing the data blocks. Similarly when any server failure happens while accessing any data block, the same procedure will be repeated to retrieve the data using other disks using the function ***CreateFailedDataOnGet(ph_block_number,block_server_id).***

- ## PUT()

The PUT method is exposed to the client by the server to modify any block and update respective checksum data. The below **pseudo code shows the PUT code** used by the client.

```
def Put(self, virtual_block, block_data):
    if virtual_block in range(0,TOTAL_NUM_BLOCKS):
                    #converting the raw data into byte array to size block size
          putdata = bytearray(block_data.ljust(BLOCK_SIZE,b'\x00'))
                     #Calculate the physical block number , data block server id and parity server id
                     for a given virtual address
          ph_block_number,block_server_id,parity_server_id= self.MappingFunc(virtual_block)
     try:
        if block_server_id <= Total_Num_Servers:
              #updating the parity before putting the actual data to the block.
           self.UpdateParity(ph_block_number,block_server_id,parity_server_id,putdata
             #putting the actual data to the physical block in the respective server
           ret = self.block_serverN.Put(ph_block_number,putdata)
           if ret == -1:
              logging.error('Put: Server returns error')
              quit()
           return 0
       else:
          logging.error('Put: Block out of range: ' + str(virtual_block))
          quit()
     except ConnectionRefusedError:
           #if connection refused error happen on PUT
       Logging.error("Connection error while updating the data-  updated the respective parity")
        Pass
```

For every PUT operation the parity should be updated first because the old data will still be available in the server to compute the new parity. New parity can be compute using the below equation.

New_Parity = Old_data XOR New_data XOR Old_parity

The pseudo code to compute and update new parity before every PUT operation is below,

```
def UpdateParity(self,ph_block_number,block_server_id,parity_server_id,new_data ):
                    #initializing
      old_data = bytearray(BLOCK_SIZE)
      old_parity  = bytearray(BLOCK_SIZE)
      try:
         if block_server_id:
              #Get the old data from the data block server in the same stripe
            old_data = self.block_serverN.Get(ph_block_number)
            if old_data == -1:
                    old_data = self.CorrectDataBlock(block_server_id,ph_block_number)
      except ConnectionRefusedError:
           old_data = self.CreateFailedDataOnGet(ph_block_number,block_server_id)
```

```
        try:
            if parity_server_id:
                #Get the old parity from the parity block server in the same stripe
                old_parity = self.block_serverN.Get(ph_block_number)
            if old_parity == -1:
                old_parity = self.CorrectDataBlock(parity_server_id,ph_block_number)
        except ConnectionRefusedError:
            old_parity = self.CreateFailedDataOnGet(ph_block_number,parity_server_id)

            #New parity = Newdata XOR olddata XOR oldParity
        intermediate_result =  bytes([aA ^ b for a, b in zip(bytes(new_data),bytes( old_data))])
        new_parity = bytearray(bytes([aa ^ bb for aa, bb in zip(bytes(intermediate_result),
bytes(old_parity))]))

        try:
            #Putting the new parity
            ret = self.block_servern.Put(ph_block_number,new_parity)
            if ret == -1:
                logging.error('Put: Server returns error')
                quit()
        except ConnectionRefusedError:
            logging.error("Connection error while updating the parity")
            pass
```

- ## Corruption of data:

In this file system, any single data block can be corrupted by using argument ***(-cblk)*** passing it to the ***memoryfs_server.py*** while starting of the server. There are two ways to corrupt any given block.

By storing incorrect data:

When any put data request comes, incorrect data can be written(always) instead of the correct data. The checksum of this block will be updated with the correct hash associated with the correct data.

By storing the incorrect checksum:

When any put data request comes, incorrect will be generated and written(always) instead of the correct checksum. The data of this block will be updated correctly.

Pseudo code for corrupting the block with wrong data: (Implemented in the server PUT)

```
If   block_number == CBLK:
        #this block will always be stored with the wrong value.
      stringbyte = bytearray("Block_Corrupted","utf-8")
        #converting the wrong data into byte array of size block size
      corrupt_data = bytearray(stringbyte.ljust(BLOCK_SIZE,b'\x00'))
        #Storing the corrupted data to that block
      RawBlocks.block[CBLK] = corrupt_data
else:
        # else store the correct data
        RawBlocks.block[block_number] = bytearray(data.data)
```

This implementation will always write *"Block_Corrupted"* data to the corrupted block irrespective of the data request coming to the block. On every get request for this block there will always be mismatch between the data and checksum. So, the correct data will always be generated using the other data blocks/parity of other servers in the same physical address stripe. (Note this blocks stripe parity will always be updated correctly every time as the parity gets updated first before the actual put).

- ## Fail-stop of the server:

Fail-stop is a scenario when any server is down/temporarily unavailable for a client access. This can be emulated by bringing down the active server manually in the terminal (by pressing ctrl+c). In this scenario the client will continue its operations without failing. The main change from the normal operation is that whenever the put operation goes for this server, it will simply update the parity of the respective physical address stripe of the disks. This scenario will be corrected using the repair mechanism.

- ## Repair

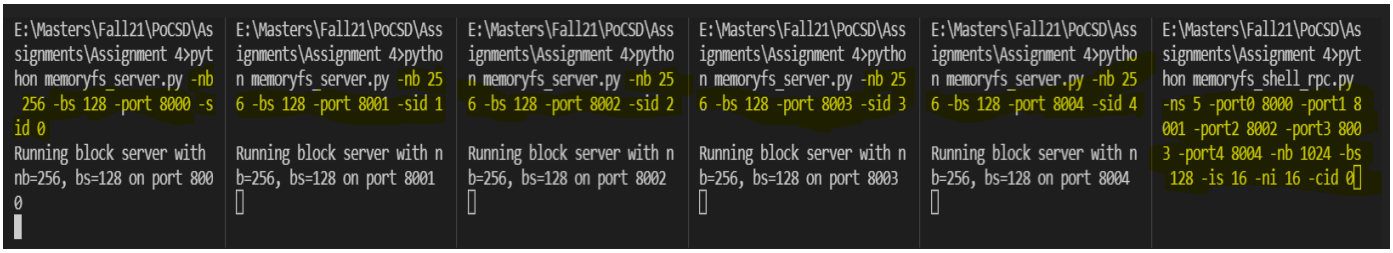
Whenever the fail-stop scenario happens, the server needs to be brought up to its normal operation. So, during the pair process, the faulty server should be run which initializes all the data blocks with zero-byte arrays. To preserve the normal operation the earlier data needs to be restored. This data restoration should be performed from the client by passing the repair flag with server id **(repair Server_id)** to ***memoryfs_shell_rpc.py.*** During the repair process, the data from the other servers of the same physical stripe will be retrieved in a loop and the XOR to get the failed server data. Below is the pseudo code for the repair process.

```
def repair(self,server_id):
    try:
     for block_num in range(TOTAL_NUM_BLOCKS//( Total_servers -1)):
          for i  in range(Total_servers):
                if server_id != i:
                        #get the data from the other servers
                       data = self.block_serverN.Get(block_num)
                 if data == -1:
                        logging. Debug("Correcting block: Unknown error")
```

```
                quit()
            corrected_data =  bytes([a ^ b for a, b in zip(bytes(corrected_data),
            bytes(data))])
              #put the data in the failed server
              ret = self.block_serverN.Put(block_num,bytearray(corrected_data))
            if ret == -1:
                quit()
        except Exception as e:
          logging. Error("Encountered two faults in the system...")
          quit()
```

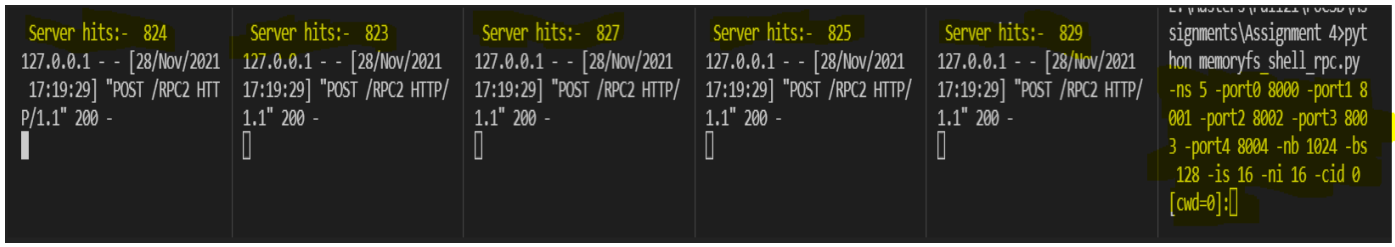
Evaluation:

The evaluation is performed with five servers (ns =5), 256 blocks on each server (nb = 256), 128 bytes block size (bs = 128), number blocks passed from the server is 1024.



Initial evaluation Setting

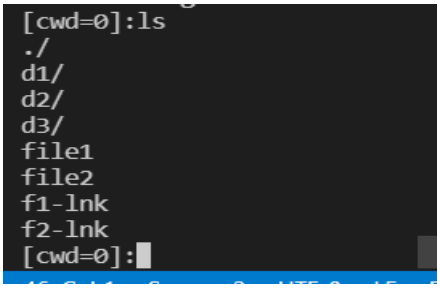
When the client shell runs all the blocks will be configures according to the RAID5 mapping function with one parity block on each strip of physical address. Any multiple requests will be shared across all the servers accordingly as the blocks are spread across the n servers.



Initial server hits to the servers

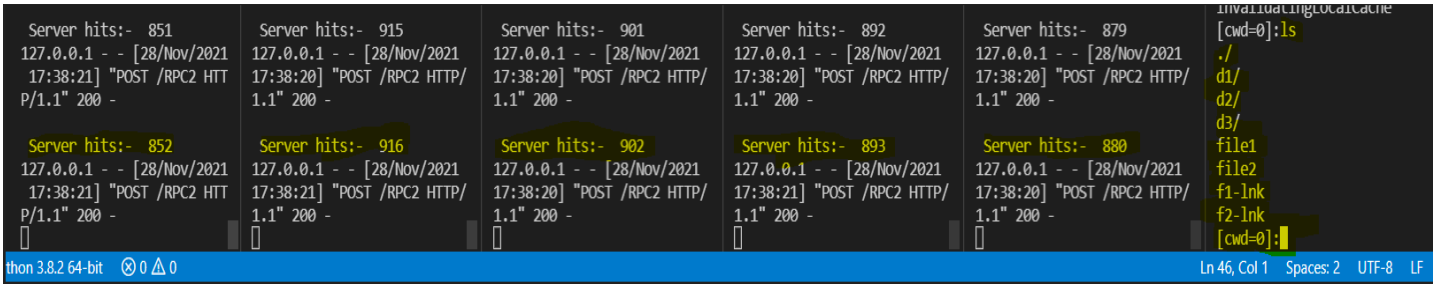
After client runs the above screen shot shows the number of requests handled by the server is printed. From the displayed server hits is evident that no sever is a hotspot for a client request.

The further evaluation is performed by creating the simple file structure from the shell as shown below. The file system has three directories two files and two hard links to these files.



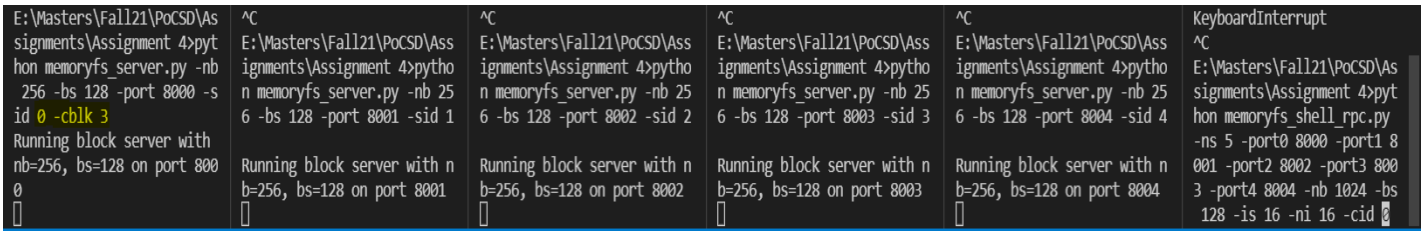
The file structure created for 9valuation

After creating this file structure, there has been many server requests generated by the client to get the data and put the data from the disk block. Below figure shows the number of accesses after these operations.

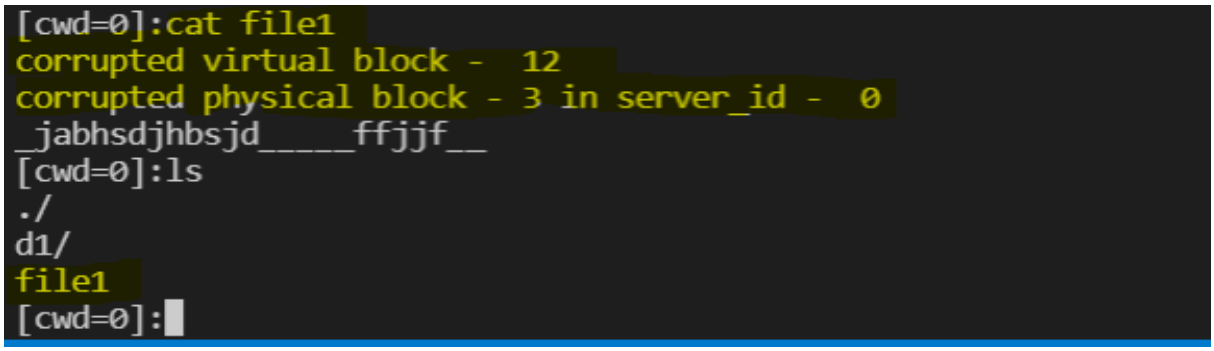


No of hits for the servers after creating the file system.

The corrupt block scenario is tested in the **server 0 with physical block 3**(which is the virtual block 12) which is the stating block for the data block in this testing configuration. Below figure shows the command window before executing the corrupt block scenario.

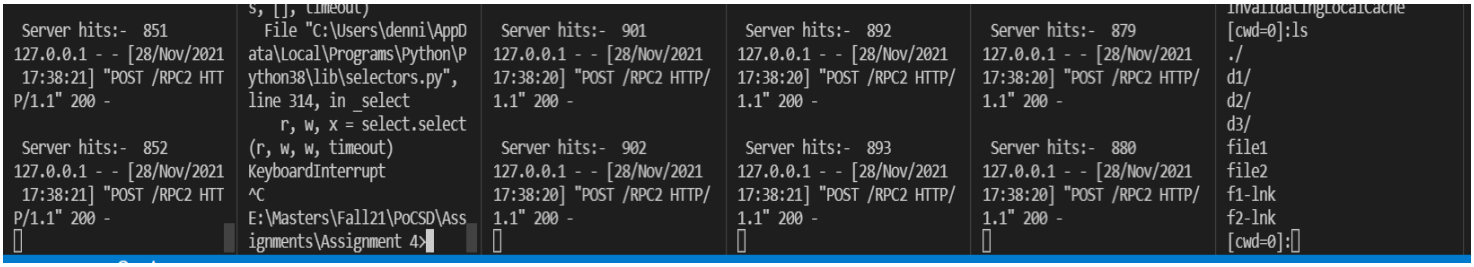


Corrupting the block 3 in server 0 (data block)



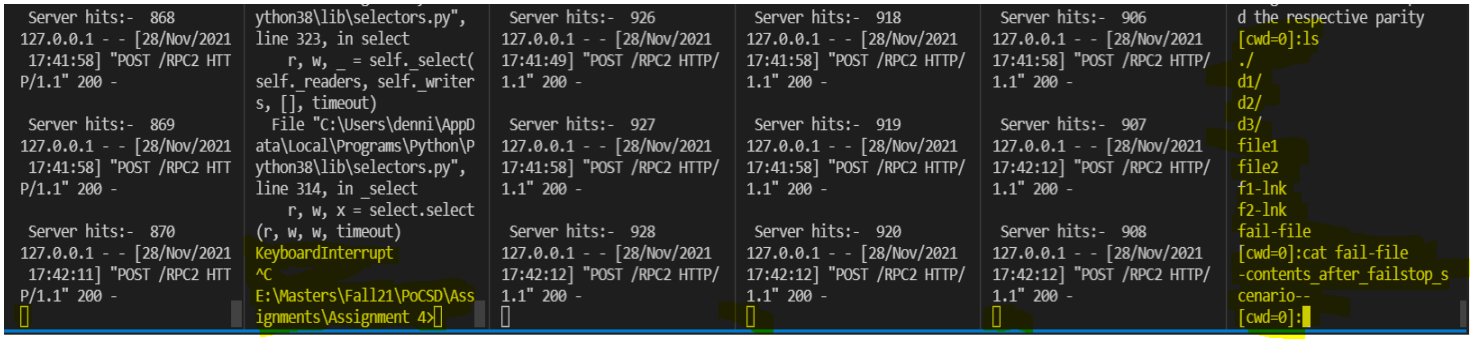
Shell operation after corrupting the block

After corrupting the first data block (virtual block number 12) when the data access (get request to this block) there will be checksum mismatch then the get operation will recreate the original data by XORing the other data blocks in the same stripe. The above screen shot shows this scenario, when the CAT is executed, the request goes to 12th block which result in the checksum error the correct contents are restored using the other data block in the other servers using the function ***def CreateFailedDataOnGet(self,ph_block_number,server_id)*** in ***memory_client.py.***



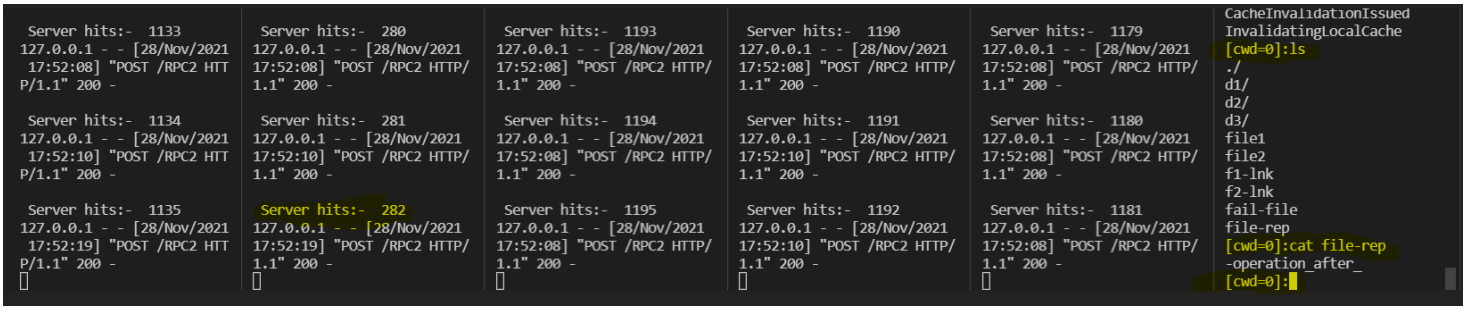
Fail-stop of server 1

The **Fail-stop scenario** is tested on the **server1** as shown in the above snapshot. At this point the client is not crashed as connection refused error is being handles for every access to server1. Now what ever the operation happen the client should work without any failure.



Successful shell operation after fail-stop

As we can see in the above console while the server1 is down the CAT command on the client is success in retrieving the contents and displayed (the process of retrieving as described above). The next step would be to bring server 1 up and test the normal operation of the system after repair process.

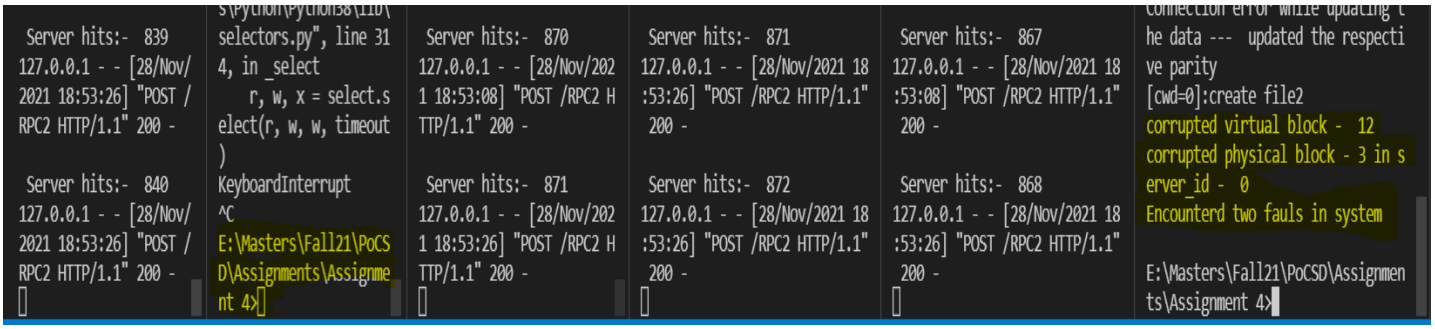


Successful shell operation after repair of server 1

Post repair of the server1, by seeing the number of requests served by server1(282) it is very clear that server1 retains its content from the other disks and continue performing its operation. The file creation, append and CAT operation is success after repair.

The scenario with two faults in the system – when the corrupt block **(corrupted block 3 in the server0)** and the fail-stop **(of the server1)** faults are present in the system, this system design does not tolerate any further operation on the servers. Eventually it failed to stop when the system encounters this scenario. Below snapshot shows failing of the system automatically in this scenario. The client will print "Encountered two faults in the system" then it automatically quits.

Two failures in the system (corrupt block, fail-stop)

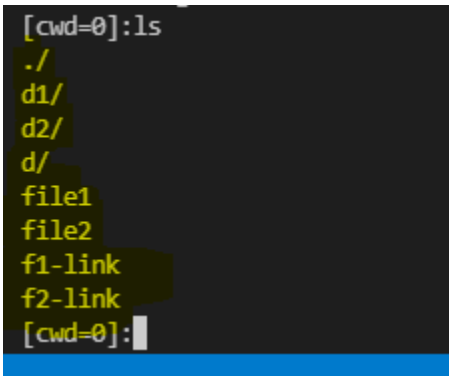


- ## Performance:

The performance measurement is the key factor for any system design. There are many reasons that multi server is preferred over single server system. The biggest reason is backup. In case e server fails then data is fetched from other disks. Another important reason is to reduce the load on each server which increase the response time of the system. This project is designed to support multiple servers up to 8 servers with one client. When any client performs any operation/request the server the requests will be distributed equally among N servers.

The average load on each server = $\frac{Total\ No.of\ Requests}{Total\ no.of\ servers}$

The performance evaluation is conducted on single server system and five server system by creating the following operations from the client.



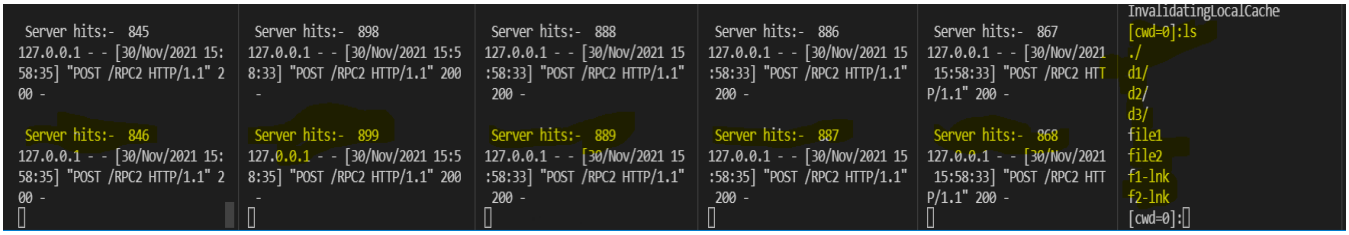
Note: *In this setup there are few implementation differences between single server system and multi-server system. Single server system has the locking mechanism on every server access and no checksum implementation (which eliminates the significant requests to the server). Whereas the multi-server system has no locking mechanism, and checksum implementation is present. For very get and put access the checksum requests are additional overhead in this multi-server system. So, there won't be any proportional relation in the result.*

Performance with 1024 blocks with 128 bytes block size:

When the client is executed with 1024 blocks with each block of size 128 bytes each server has the following number of requests.

For a multi-server system:

Server	No. of requests
Server 0	846
Server 1	899
Server 2	889
Server 3	887
Server 4	868



For a single server system:

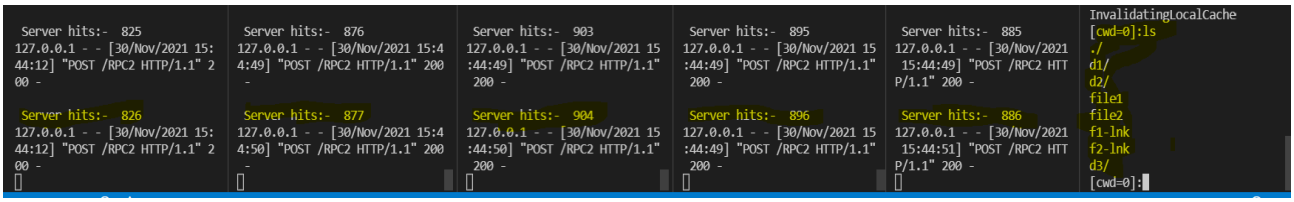
Server	No. of requests
Server 0	1156

Performance with 1024 blocks and 256 bytes block size:

When the client is executed with 1024 blocks with each block of size 256 bytes each server has the following number of requests.

For a multi-server system:

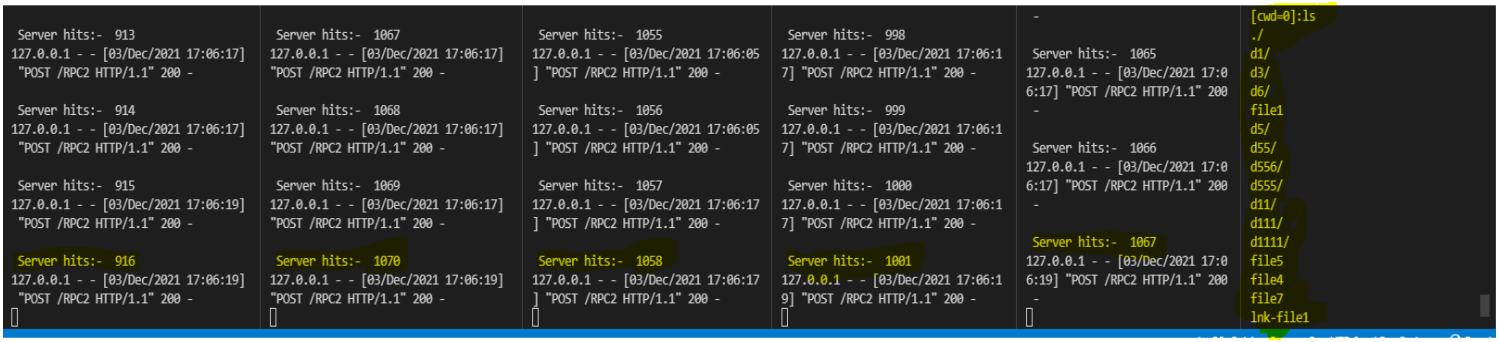
Server	No. of requests
Server 0	826
Server 1	877
Server 2	904
Server 3	896
Server 4	886



For a single server system:

Server	No. of requests
Server 0	1184

Performance with 1024 blocks and 128 bytes block size but bigger file sizes:



The load on the servers appears to be spread, and no single server appears to be a hotspot for client requests.

It is observed that for the same client operation, for multi-server system the load on each server is decreased by almost **30%,** with a single server system.

- ## Reproducibility

The following steps helps to run the evaluation of the scripts. (python should be installed in the system as prerequisite to execute these commands),

1. Navigate to the python scripts directory and open the command windows as many as number of server and clients in the system to test. (In this testing, no of servers =5 and clients =1, so total 6 command windows),
2. To run the server, execute the following command

   python memoryfs_server.py -nb 256 -bs 128 -port 8000 -sid 0
3. Repeat this step to run the other servers by changing the port and server id.
4. To execute the client, execute the following command then the shell interpreter will run on the window.

   python memoryfs_shell_rpc.py -ns 5 -port0 8000 -port1 8001 -port2 8002 -port3 8003 -port4 8004 -nb 1024 -bs 128 -is 16 -ni 16 -cid 0
5. Specify all the active server ports and the total number of blocks. If no. of servers N=5 with 256 blocks each, then client should be started with 1024 blocks because there are (N-1)*256 data blocks available,
6. To corrupt block in any server, the server should start with the following command before running the client.(in this test block number 3(physical block) in the server is corrupted),

   python memoryfs_server.py -nb 256 -bs 128 -port 8000 -sid 0 -cblk 3
7. To fail stop any server, press ctrl+c on the command window of any running server.
8. To repair any failed server, execute the following command in the shell (if server 0 is failed).

   repair 0
9. To stop any server or client operation press ctrl+c on the respective console window.

Conclusions

- This project implemented multi-server single client-based file system where clients accessed data blocks stored at a server.
- Design of systems including: client/service, networking, and fault tolerance with support of Redundant Array of Independent Disks (RAID5) block storage approach.
- The Mapping function in the system can be adjusted to work the system like any other RAID versions (like RAID 4, RAID5…)
- This files system is implemented using checksum which helps to identify any errors in the system and this design doesn't include the RSM locks mechanism.
- Designed is robust enough to handle and mitigate the failure scenarios like, Corrupt block (only one) and Fail-stop of server.
- The repair mechanism is implemented to bring the server back with its old contents to which resumes normal operation.
- This file system design cannot handle more than two failures in a system at any given time.
- It is observed that, for multi-server system the load on each server is decreased by almost **30%,** with a single server system (allowing few implementation changes between two servers).