# RAID-5 STORAGE AREA NETWORK

## 1. INTRODUCTION AND PROBLEM STATEMENT

With rapid increase in data from last two decades, the effective and dependable data storage systems are imperative . Our project titled "RAID-5 STORAGE AREA NETWORK" (RAID-5 SAN) aims to address this need by emulating a distributed fault tolerant storage solution on a python based UNIX file system. This system utilizes a client/server model with Remote Procedure Calls (RPC) that facilitate network interaction while promoting server as dependable and efficient storage system.

The primary challenges addressed by our project involve uneven load distribution and fault tolerance, common in traditional systems, which significantly impact performance and reliability. We specifically target two types of failures: Soft Failure, wherein data degradation occurs over time, and Hard failure, wherein complete server shutdown takes place. To confront these challenges, our system embraces RAID-5 configuration, utilizing checksums and distributed parity to enable data recovery from corruption and provide repair capabilities upon disk replacement or failure.

## 2. DESIGN AND IMPLEMENTATION

This section delves into the design solutions proposed to tackle the client's capability to interface with several storage servers via RPC, the servers' capacity to leverage checksums for detecting corrupted data blocks, and the employment of a distributed parity system to facilitate data recovery and repair processes for disk failures.

### 2.1 MULTI-SERVER CONNECTION

For the client's capability to link with numerous servers, we included two supplementary arguments into the **fsmain.py** file - **'-startport'** to denote the initial port and **'-ns'** to indicate the overall number of servers. This addition assumes that each server possesses a distinct port ID commencing from the specified **'startport'**.

```
ap.add_argument('-ns', '--number_of_servers', type=int, required=True, help='the number of servers')
ap.add_argument('-startport', '--start_port', type=int, required=True, help='the starting port for the first server')
```

Additionally, the **fsconfig.py** file is modified to retain these parameters for each client. Subsequently, the details derived from **fsconfig.py** are employed by **block.py** to establish an XMLRPC connection to the designated storage servers. Moreover, a hash map named **block_servers** is implemented at the client-side to accommodate the proxy objects of the RPC servers, allowing for the invocation of server methods.

```
self.block_servers = []

# initialize XMLRPC client connection to raw block server
if fsconfig.START_PORT:
    START_PORT = fsconfig.START_PORT
else:
    print('Must specify port number')
    quit()

if fsconfig.MAX_SERVERS:
    MAX_SERVERS = fsconfig.MAX_SERVERS
else:
    print('specify total number of servers > 2')
    quit()

for i in range(MAX_SERVERS):

    PORT_ID = START_PORT + i
    server_url = 'http://' + fsconfig.SERVER_ADDRESS + ':' + str(PORT_ID)
    block_server = xmlrpc.client.ServerProxy(server_url, use_builtin_types=True)
    self.block_servers.append(block_server)
    socket.setdefaulttimeout(fsconfig.SOCKET_TIMEOUT)
```

To create a uniform disk appearance for the distributed system, we developed a mapping function known as **virtual_to_physical**, which links the client-side virtual block numbers to their corresponding block numbers located on the designated server. Additionally, this function identifies the position of the parity block associated with the virtual block number. Based on the RAID-5 striping algorithm, the virtual_to_physical method disperses data across various disk servers, including an extra parity block for fault tolerance, a concept explored further in forthcoming sections. Since one block out of each set is utilized for storing the parity, the formula **actual_block_number = virtual_block_number % NumServers - 1** is applied to calculate the actual block number. The **block server identifier** is determined using = **virtual_block_number // NumServers - 1** (equivalent to set id) . The parity_id denotes the server ID holding the parity block = (NumServers -1) - ((parity_block_num) % (NumServers). This calculation effectively distributes the parity across diverse servers. To confirm that the parity server ID does not match the block server ID, if the parity ID is less than or equal to the block server ID, we increment the block server ID by 1.

```
def virtual_to_physical(self,virtual_block_numb):

    # Generate Parity information
    NumServers = fsconfig.MAX_SERVERS
    parity_block_numb = virtual_block_numb // (NumServers-1)
    # effective roation of parity across different server in
    # anticlockwise direction
    parity_id = (NumServers -1) -((parity_block_numb) % (NumServers))

    # Generate physical block locations
    block_server_index = virtual_block_numb % (NumServers-1)
    actual_block_num = (virtual_block_numb // (NumServers-1))

    #Check for indexing issues
    if parity_id <= block_server_index:
        block_server_index = block_server_index + 1


    return block_server_index,actual_block_num,parity_id,parity_block_numb
```

## 2.2 CORRUPTION OF DATA BLOCK AND CHECKSUM

Data block corruption arises when a specific block becomes damaged due to external factors or natural deterioration over time. To illustrate this phenomenon in our emulation, we utilize the **'-cblk <block_number>'** command argument within the **blockserver.py,** where the **<block_number>** represent the actual block present within the server itself. When a particular server with corrupted block is initialized it is specified at the server side.

```
ap.add_argument('-cblk','--corrupted_block',type = int,help='specific block in the block array an interger value')
```

```
CORRUPTED_BLOCK = -1
if args.corrupted_block is not None or args.corrupted_block == 0:
    CORRUPTED_BLOCK = args.corrupted_block
    print("CORRUPTED_BLOCK: " + str(CORRUPTED_BLOCK))
```

To rectify the effects of corrupted blocks and preserve data integrity at the server level, we implement a hash map **'checksums'** storing the respective block's checksum. This checksum is computed using the 128-bit MD5 algorithm available in the hashlib Python library.

```
def md5_checksum(data):

    md5 = hashlib.md5()
    md5.update(data)
    return md5.hexdigest()
```

Further modifications were introduced to **blockserver.py** to accommodate the data block corruption. Within the **Get()** function, if the assessment determines the block to be corrupted, the client receives a response of -1. Simultaneously, **"CORRUPTED BLOCK <ACTUAL BLOCK NUMBER>"** is displayed at the server end, whereas **"CORRUPTED BLOCK <VIRTUAL BLOCK NUMBER>"** is exhibited at the client-level. Upon examination of the **Put()** function, if the **block is detected as corrupted**, the procedure allocates a fixed checksum, represented by the **CORRUPT_DATA** byte array variable, for the **block's checksum retention**. Rather than saving the original data, the **CORRUPT_DATA_2** byte array variable, containing alternative content, is stored t**o avoid the possibility of having identical checksums** between the data block and the preserved checksum within the hash map.

```
#corrupted_data if corrupted_block
temp_data = bytearray("BLOCK_CORRUPTED", "utf-8")
CORRUPT_DATA = bytes(temp_data.ljust(BLOCK_SIZE,b'\x00'))

temp_data = bytearray("BLOCK_CORRUPTED_SHOW_MESSAGE_BLOCK", "utf-8")
CORRUPT_DATA_2 = bytes(temp_data.ljust(BLOCK_SIZE,b'\x00'))
```

```
def Get(block_number):

  global SERVER_HITS
  SERVER_HITS = SERVER_HITS + 1
  result = RawBlocks.block[block_number]
  checksum = md5_checksum(result)

  if checksum != RawBlocks.checksums.get(block_number,None):
    #print('the result value ',result)
    print("BLOCK_CORRUPTED :" +str(block_number))
    RawBlocks.Sleep()
    return -1

  RawBlocks.Sleep()
  return result
```

```
#put with MD5 checksum
def Put(block_number, data):

  global SERVER_HITS
  SERVER_HITS = SERVER_HITS + 1
  RawBlocks.block[block_number] = data.data
  checksum = md5_checksum(RawBlocks.block[block_number])

  if block_number == CORRUPTED_BLOCK:
    checksum = md5_checksum(CORRUPT_DATA)
    # no point of this because we send the corrected data
    # to the output even after
    # storing the corrupt data block message.
    # This is because of recoverblock
    # we cant print the block corrupted show message.
    RawBlocks.block[block_number] = CORRUPT_DATA_2

  RawBlocks.checksums[block_number] = checksum
  RawBlocks.Sleep()
  return 0
```

```
def RSM(block_number):

 global SERVER_HITS
 SERVER_HITS = SERVER_HITS + 1
 RSM_LOCKED = bytearray(b'\x01') * 1
 result = RawBlocks.block[block_number]
 # RawBlocks.block[block_number] = RSM_LOCKED
 #since adding new value to the block need to update the checksum.
 inital_checksum = md5_checksum(result)
 if inital_checksum == RawBlocks.checksums[block_number]:
   put_data = bytearray(RSM_LOCKED.ljust(BLOCK_SIZE,b'\x01'))
   RawBlocks.block[block_number] = put_data
   checksum = md5_checksum(put_data)
   RawBlocks.checksums[block_number] = checksum
   RawBlocks.Sleep()
 return result
```

Lastly, upon examining the **RSM() function**, wherein the final block serves as a thread lock mechanism, we guarantee that the checksum verification process confirms the absence of corruption before proceeding with the lock acquisition. Should the preceding criterion be satisfied, the server obtains the lock, modifying its state to **RSM_LOCKED** while simultaneously **updating the associated checksum** value.

## 2.3 DISTRIBUTED PARITY SYSTEM FOR FAULT TOLERANCE

In order to ensure data reliability and integrity , numerous changes were implemented within the **Put()** and **Get()** method in **block.py**. Besides, **SimplePut()** and **SimpleGet()** operations were also defined, which invoke the corresponding RPC methods for the blocks in the designated servers. Additionally, the **RecoverBlock()** and **Repair()** methods were constructed to recover data for faulty blocks and server. An exhaustive explanation of these methods follows.

```
def SingleGet(self,server_id,block_number,virtual_block_number):

    if(server_id!=self.bad_server):
        try:
            data = self.block_servers[server_id].Get(block_number)
            #print(f'dat from single get {data}')
        except (socket.timeout, ConnectionRefusedError) as e:
            self.bad_server = server_id
            data = self.RecoverBlock(server_id,block_number,virtual_block_number)
            print(f"SERVER_TIMED_OUT due to {e} for server {server_id}")
            print(f'SERVER_DISCONNECTED GET {virtual_block_number}')
            time.sleep(fsconfig.RETRY_INTERVAL)
    else:
        print(f'SERVER_DISCONNECTED GET {virtual_block_number}')
        data = self.RecoverBlock(server_id,block_number,virtual_block_number)
        print('block recovered')
    return data
```

The **SimpleGet()** method accepts the block number and its respective server ID as input arguments along with the virtual block number. Using a try-catch approach with the at-most-once semantic, a RPC call is initiated towards the **Get()** method on the specified server ID to retrieve data from the particular block. If the function call succeeds then retrieved data is forwarded to the calling function. Otherwise, the exception is raised, the disconnected server ID is stored and the message is logged - **“SERVER_DISCONNECTED GET <VIRTUAL BLOCK NUMBER>**”. Concurrently, the data recovery function **RecoverBlock** , detailed in subsequent discussion, is invoked to recuperate the actual data. Prior to executing these steps, the **SimpleGet() function** evaluates wether the provided server ID is flawed. In case of a defective server, similar actions described under the exception clause are executed. Thus, the SimpleGet() function manages the hard failure related to server malfunction.
In addressing soft failures resulting from corrupted blocks, the **SimpleGet()** method communicates a return value of **'-1'** (obtained from the server) to the function caller, denoting the presence of a corrupted block. This action triggers the presentation of a message stating **"BLOCK CORRUPTED <VIRTUAL BLOCK NUMBER>",** followed by the application of the data recovery function to recreate the accurate block.

```
def SinglePut(self,server_id,block_number,put_data_block):

    if server_id!=self.bad_server:
        try:
            ret = self.block_servers[server_id].Put(block_number,put_data_block)
        except (socket.timeout, ConnectionRefusedError) as e:
            print(f"SERVER_TIMED_OUT due to {e} for server {server_id}")
            self.bad_server = server_id
            print(f'SERVER_DISCONNECTED PUT {server_id}')
            ret = -1
            time.sleep(fsconfig.RETRY_INTERVAL)

    else:
        ret = -1

    return ret
```

Similar to the SingleGet() method, the **SinglePut()** method accepts server ID, block number, and additionally the data block intended for storage. Implementing the **Put() method** with the same at-most-once approach, it yields a return value of 0 upon successful execution; otherwise, -1 is returned, accompanied by the storage of the faulty server ID and the display of the message **"SERVER_DISCONNECTED PUT <SERVER_ID>".** Before reaching this stage, if the provided server ID is deemed faulty, the method returns -1 without making an RPC call to the Put() method. If the calling function encounters a return value of **"-1"**, it prompts the message **"SERVER DISCONNECTED PUT {VIRTUAL BLOCK NUMBER}".**

```
def RecoverBlock(self,server_id,block_number,virtual_block_number):

    put_data = bytearray(fsconfig.BLOCK_SIZE)
    for i in range(fsconfig.MAX_SERVERS):
        if i!=server_id:
            server_data = self.SingleGet(i,block_number,virtual_block_number)
            put_data = bytearray(self.xor_blocks(put_data,server_data))
    return put_data
```

The **RecoverBlock()** function plays a vital role in restoring the data of a faulty block. Initially, an empty byte array named **put_data** is produced. Through iterative progression, this array is XORed with data blocks belonging to the same set on the remaining servers. The xor_blocks() method used to XOR the two blocks is represented below

```
def xor_blocks(self,b1,b2):

    xor = bytearray()
    for a1,a2 in zip(b1,b2):
        xor.append(a1^a2)

    return xor
```

```
data_id,data_block,parity_id,parity_block = self.virtual_to_physical(block_number)
data = self.SingleGet(data_id,data_block,block_number)
#if(block_number == fsconfig.TOTAL_NUM_BLOCKS-2):
    #print(data)
    #print(f'data id {data_id} data_block {data_block} parity_id {parity_id} parit

#checking if data is corrupt or not
if data == -1:
    print(f'CORRUPTED_BLOCK {block_number}')
    data = self.RecoverBlock(data_id,data_block,block_number)
# add to cache
self.blockcache[block_number] = data
```

To modify the **Get()** method for a multi-server system, certain steps are undertaken. Initially, the **virtual_to_physical()** function identifies the actual data block with its server ID, and the associated parity block with their respective server IDs, accepting virtual block numbers as input. Next, the **SimpleGet()** function is used, providing the actual data block and its server ID as arguments.This function efficiently handles hard and soft failures. The Get() method offers extra protection against soft failures by creating a recovery block when receiving a return value of -1, as illustrated in the SimpleGet() function. Lastly, the obtained data block, either from the SimpleGet() method or the RecoverBlock() function, is cached on the client-side, indexed by its corresponding virtual block number.

Modifying the **Put()** method involved more effort than adjusting the **Get()** method. Initially, similar to the Get() method, the respective data blocks and parity blocks along with their server IDs were established. A new **put_data** block was formed to store the newly computed parity in the parity block belonging to the set of the data block. Due to the necessity of updating the parity block after any modification in a particular set's block data, the new parity was calculated by performing the XOR operation on the old data block, old parity block, and new data block. The old data block and old parity block were found using a fault-tolerant approach equivalent to the one explained in the SimpleGet() function. Following the computation of the parity block, both the parity block and data block were updated with the fault tolerance implemented in the SimplePut() method. Additionally, the new data was cached on the client-side, referenced by its virtual block number. Finally, similar alterations were incorporated into the block of code responsible for updating the most recent client ID that caches the distributed server.

For the existing methods such as **RSM()**, **Acquire()**, and **Release()** originally designed for a singular server storage system, only minimal adjustments were necessary. Here, the primary modification involved incorporating the virtual_to_physical method to transform virtual block numbers into physical data blocks.

```
#first find the physical server and block for data and parity
data_id,data_block,parity_id,parity_block = self.virtual_to_physical(block_number)

#data for block
put_data_block = bytearray(block_data.ljust(fsconfig.BLOCK_SIZE,b'\x00'))
# if(block_number == fsconfig.TOTAL_NUM_BLOCKS-2):
#     print(put_data_block)

#data for parity # caution need to add if the server is present or not
old_data_block = self.SingleGet(data_id,data_block,block_number)
#check if old_data_block is not corrupted
if old_data_block == -1:
    print(f'CORRUPTED_BLOCK {block_number}')
    old_data_block = self.RecoverBlock(data_id,data_block,block_number)
old_parity_block = self.SingleGet(parity_id,parity_block,block_number)
#check if old_parity_block is not corrupted
if old_parity_block == -1:
    print(f'CORRUPTED_BLOCK {block_number}')
    old_parity_block = self.RecoverBlock(parity_id,parity_block,block_number)

#print(old_data_block)
#print(old_parity_block)
im_parity_block = self.xor_blocks(old_data_block,put_data_block)
new_parity_block = self.xor_blocks(im_parity_block,old_parity_block)

#store data # caution need to add if the server is present or not
if self.SinglePut(data_id,data_block,put_data_block) == -1:
    print(f'server down {data_id}')
    print(f'SERVER_DISCONNECTED PUT {block_number}')
if self.SinglePut(parity_id,parity_block,new_parity_block) == -1:
    print(f'server_down {parity_id}')
    print(f'SERVER_DISCONNECTED PUT {block_number}')

# update block cache #need to add conditon for showing cache.
if fsconfig.CACHE:
    print('CACHE_WRITE_THROUGH ' + str(block_number))
self.blockcache[block_number] = put_data_block
```

To restore the contents of the restored failed disk linked to the distributed file system, a **RepairServer()** method was formulated. Accepting the server ID as an argument, it compares this value with the actual faulty server held within the **bad_server** variable. If they correspond, an empty block is produced, and the **RecoverBlock()** method is employed with the server ID and block number to recuperate the data. **The retrieved data block is then saved back onto the same server ID**. This process is repeated for all the blocks present in the storage server. Once completed, a log entry is recorded indicating **"Server Repaired <Server _ID>"**. To facilitate

this method on the command line emulation of the UNIX file system following block of code was added to the shell.py file.

```
def RepairServer(self,server_id):

    if server_id == self.bad_server:
        self.bad_server = -1
        num_server_blocks = fsconfig.TOTAL_NUM_BLOCKS // (fsconfig.MAX_SERVERS-1)
        for block in range(num_server_blocks):
            put_data_block = self.RecoverBlock(server_id,block,0) # here 0 cause
            self.SinglePut(server_id,block,put_data_block)
        print(f'Server Repaird {server_id}')
```

```
elif splitcmd[0] == "repair":
    if len(splitcmd)!= 2:
        print("Error: repair requires 1 server ID")
    else:
        self.RawBlocks.Acquire()
        self.repair(splitcmd[1])
        self.RawBlocks.Release()
```

```
#implements server repair
def repair(self,server_id):
    self.RawBlocks.RepairServer(int(server_id))
    return 0
```

To assess load distribution, additional methods were incorporated into **blockserver.py**, **block.py**, and **shell.py**. Initially, in **blockserver.py**, a global variable **SERVER_HITS** was initialized to keep track of the **frequency of RPC calls** made to the server. For each RPC method call relating to **GET()**, **PUT()**, or **RSM()**, the **SERVER_HITS** counter was increased by 1. Furthermore, a function named **ServerLoad()** was developed, returning the SERVER_HITS count to the client.

```
def ServerLoad():
    global SERVER_HITS
    SERVER_HITS = SERVER_HITS + 1
    return SERVER_HITS
```

Subsequent to defining the SERVER_HITS, the **ShowLoad()** method was implemented on the client-side. This function issued an RPC call to **ServerLoad()** for every accessible server. If the server was operational, the message **"SERVER[<SERVER ID>]:requests=<SERVER HITS>"** was presented. If the server was unavailable, the output included **"SERVER[ <SERVER ID> ]:requests=0"** combined with **"SERVER DISCONNECTED ShowLoad".** Similarly, if the server ID corresponded to a flawed server ID, analogous information was conveyed. Eventually, the cumulative requests placed by clients along with the mean number of server requests processed were displayed.

```python
#get the total number of hits, individual hits and average server load
def ShowLoad(self):

    total_hits = 0
    for i in range(fsconfig.MAX_SERVERS):
        if i!=self.bad_server:
            try:
                server_hit = self.block_servers[i].ServerLoad()
                print(f'Server [{i}] requests = {server_hit}')
            except (socket.timeout, ConnectionError, ConnectionRefusedError) as e:
                server_hit = 0
                print(f'Server [{i}] requests = {server_hit}')
                print(f"SERVER_TIMED_OUT due to {e} for server {i}")
                print(f'SERVER_DISCONNECTED ShowLoad')
                time.sleep(fsconfig.RETRY_INTERVAL)
        else:
            server_hit = 0
            print(f"SERVER_TIMED CONNECTED_ERROR for server {i}")
            print(f'SERVER_DISCONNECTED ShowLoad')
            time.sleep(fsconfig.RETRY_INTERVAL)
        total_hits = total_hits + server_hit

    print(f'The total server requests = {total_hits}')
    average_hits = total_hits//fsconfig.MAX_SERVERS
    print(f'The average server hits = {average_hits}')

    return 0
```

```python
#implements show load
def showload(self):
    self.RawBlocks.ShowLoad()
    return 0
```

```python
elif splitcmd[0] =="showload":
    self.RawBlocks.Acquire()
    self.showload()
    self.RawBlocks.Release()
```

Finally, to facilitate the execution of this functionality in the emulated shell for the UNIX File System, the following code snippet was appended to shell.py.

The final alteration implemented in **shell.py** entailed showcasing cache-related logs on the client-side. An auxiliary flag titled **-logcache** was inserted into **shell.py**, allowing only two possible values: **0** and **1**. Here, 0 denoted suppressing the log and 1 indicated displaying the log. Subsequently, this value associated with -logcache was retained in the **CACHE** variable declared in the **fsconfig.py** file. Consequently, for each section of code where cache logging occurred, it was enclosed within an if statement as demonstrated in the **Get()** function .

```
if (block_number < fsconfig.TOTAL_NUM_BLOCKS-2) and (block_number in self.blockcache):
    if fsconfig.CACHE:
        print('CACHE_HIT '+ str(block_number))
    data = self.blockcache[block_number]
```

# 3. EVALUATION

To validate our design, we conducted tests on a system comprising four servers. Each server possessed a block size of 128 bytes and an inode size of 32 bytes, with a maximum of 16 inodes. Consequently, the system could support a maximum file size of 768 bytes. Several files ranging from f1 to f6 were initially created, containing 512 bytes of data per file. Employing the showload() function ensured the equitable distribution of load among the servers, thereby verifying the multi-network character of the storage system. Two clients connected to the storage area network were considered for this experiment; one client with cid-1 did not log cache, whereas the second client with cid-2 did.

```
#### File system information:
Number of blocks          : 384
Block size (Bytes)        : 128
Number of inodes          : 16
inode size (Bytes)        : 32
inodes per block          : 4
Free bitmap offset        : 2
Free bitmap size (blocks) : 3
Inode table offset        : 5
Inode table size (blocks) : 4
Max blocks per file       : 6
Data blocks offset        : 9
Data block size (blocks)  : 375
```

```
[cwd=0]%showload
CACHE_MISS 382
Server [0] requests = 310
Server [1] requests = 53
Server [2] requests = 384
Server [3] requests = 204
The total server requests = 951
The average server hits = 237
CACHE_WRITE_THROUGH 383
```

With data distributed across numerous blocks residing in distinct servers, efforts were directed toward closing server ID 1. Simultaneously, both clients attempted to access various files on the storage network while displaying log files and demonstrating the system's fault tolerance via recovery blocks. Upon examining the outputs of the commands cat f1 and cat f5, it became apparent that the file system successfully exhibited the contents of the files even after shutting down server ID 1. This exhibited the strong reliability and fault tolerance of the system.

Notably, the server continued to generate new files despite experiencing a server failure. Additionally, it recognized the flawed operation on the compromised server and seamlessly transferred the remaining tasks to the operable server, evidenced by the command **create f8** on

```
[cwd=0]%cat f1
CACHE_MISS 382
CACHE_HIT 5
CACHE_HIT 5
CACHE_HIT 10
CACHE_HIT 5
CACHE_HIT 5
CACHE_HIT 5
CACHE_MISS 11
CACHE_MISS 12
CACHE_MISS 13
SERVER_TIMED_OUT due to [Errno 61] Connection refused for server 1
SERVER_DISCONNECTED GET 13
CACHE_MISS 14
73a42f3cc92a28c12279d2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec453cc34ffafcd
29432d17b0afb9c78f9b3a82041aa7c5a3b483dd0a3fa0373a42f3cc92a28c12279d2db46361a6c82
39e3e03201fa8591e96aafb6b27d2aebec453cc34ffafcd29432d17b0afb9c78f9b3a82041aa7c5a3
b483dd0a3fa0373a42f3cc92a28c12279d2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec
453cc34ffafcd29432d17b0afb9c78f9b3a82041aa7c5a3b483dd0a3fa0373a42f3cc92a28c12279d
2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec453cc34ffafcd29432d17b0afb9c78f9b3
a82041aa7c5a3b483dd0a3fa03
CACHE_WRITE_THROUGH 383
```

```
[cwd=0]%cat f5
SERVER_DISCONNECTED GET 28
block recovered
73a42f3cc92a28c12279d2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec453cc34ffafcd
29432d17b0afb9c78f9b3a82041aa7c5a3b483dd0a3fa0373a42f3cc92a28c12279d2db46361a6c82
39e3e03201fa8591e96aafb6b27d2aebec453cc34ffafcd29432d17b0afb9c78f9b3a82041aa7c5a3
b483dd0a3fa0373a42f3cc92a28c12279d2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec
453cc34ffafcd29432d17b0afb9c78f9b3a82041aa7c5a3b483dd0a3fa0373a42f3cc92a28c12279d
2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec453cc34ffafcd29432d17b0afb9c78f9b3
a82041aa7c5a3b483dd0a3fa03
```

client ID 1. The anticipated log entries **"SERVER_DISCONNECTED GET 7"** and **"SERVER_DISCONNECTED PUT 7"** displayed during this process.

```
[cwd=0]%create f8
SERVER_DISCONNECTED GET 7
block recovered
server_down 1
SERVER_DISCONNECTED PUT 7
```

To illustrate the capability to recover the data of a failed server, we shall restart server ID 1. Utilizing the **repair 1** (flawed server ID) command on client ID 1, we will initiate the restoration process for server ID 1. By employing the **showload()** function prior to and post reconnection, we aim to verify the issuance of RPC requests to the reestablished server, thus confirming the storage of data within the respective blocks housed on this server. Moreover, if it were a single server based storage there would be 2500 requests. This load is almost distributed to 1/3th across

all the servers, thus decreasing server load and increasing server performance. It is 1/3th because the server was again restored also some of the servers are accessed as they might contain the inodes that are frequently accessed.

```
[cwd=0]%showload
Server [0] requests = 680
SERVER_TIMED CONNECTED_ERROR for server 1
SERVER_DISCONNECTED ShowLoad
Server [2] requests = 758
Server [3] requests = 550
The total server requests = 1988
The average server hits = 497
[cwd=0]%repair 1
Server Repaird 1
[cwd=0]%showload
Server [0] requests = 813
Server [1] requests = 129
Server [2] requests = 889
Server [3] requests = 685
The total server requests = 2516
The average server hits = 629
```

To simulate corruption within the data storage network, we shall operate four servers, including server ID 2 with a corrupt block3. Each server possesses a block size of 128 bytes and an inode size of 16 bytes, with a total of 16 inodes. Block 3 on server ID 2 corresponds to the virtually corrupt block 10. To populate distinct data blocks throughout the distributed file system, we utilize the commands create f1, create f2, append f1 <256_byte_data>, and append f2 <256_byte_data>. Inode 2 stores the contents of file f2, which consists of block 10 and block 11, discernible through the showinode 2 command.

```
#### File system information:
Number of blocks          : 384
Block size (Bytes)        : 128
Number of inodes          : 16
inode size (Bytes)        : 16
inodes per block          : 8
Free bitmap offset        : 2
Free bitmap size (blocks) : 3
Inode table offset        : 5
Inode table size (blocks) : 2
Max blocks per file       : 2
Data blocks offset        : 7
Data block size (blocks)  : 377
```

```
[cwd=0]%showinode 2
CACHE_HIT 5
Inode size   : 256
Inode type   : 1
Inode refcnt : 1
Block numbers:
10,11,
```

Upon attempting to access block 10, which displays the contents of file f2, we observe that block 10 is corrupted, confirmed by the checksum calculation on the server-side. Nonetheless, leveraging the parity data preserved within the distributed system and the **recoverdata()** function, we effectively recover the accurate data contained within the corrupt block showing tolerance against the soft-failure.

```
[cwd=0]%cat f2
CORRUPTED_BLOCK 10
73a42f3cc92a28c12279d2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec453c
c34ffafcd29432d17b0afb9c78f9b3a82041aa7c5a3b483dd0a3fa0373a42f3cc92a28c1
2279d2db46361a6c8239e3e03201fa8591e96aafb6b27d2aebec453cc34ffafcd29432d1
7b0afb9c78f9b3a82041aa7c5a3b483dd0a3fa03
```

If we try a different configuration shown below using fsconfig using 3 servers, create 4 files and store 64 bytes of data, we get the same average hits across the server which are approximately 1/3 or even 1/4 of the total request following the same increase in performance as aforementioned example.

```
Number of blocks          : 192
Block size (Bytes)        : 64
Number of inodes          : 32
inode size (Bytes)        : 16
inodes per block          : 4
Free bitmap offset        : 2
Free bitmap size (blocks) : 3
Inode table offset        : 5
Inode table size (blocks) : 8
Max blocks per file       : 2
Data blocks offset        : 13
Data block size (blocks)  : 179
```

```
[cwd=0]%showload
Server [0] requests = 94
Server [1] requests = 19
Server [2] requests = 130
Server [3] requests = 84
The total server requests = 327
The average server hits = 81
```

# 4. REPRODUCIBILITY

To reproduce the outcomes previously depicted in the evaluation, adhere to the following introductory directions before delving into specific scenarios. Begin by launching the suitable servers using a command such as **"python3 blockserver.py -nb 128 -bs 128 -port 8005"**, ensuring that the N servers are started consecutively with increasing port numbers (N>4). Proceed to initialize the intended clients capable of interacting with the server by implementing the following command: "**python3 fsmain.py -ns 4 -bs 128 -nb 384 -cid 2 -startport 8005 -is 16 -ni 16 -logcache 1**".

The quantity of blocks accessible to each client is equal to (N-1) * (nb of the server). To verify the establishment of connections within a distributed storage network, apply the **showload()** function, enabling visualization of load dispersion across the system. This function may also serve to scrutinize load distribution among the servers. To imitate block decay, incorporate the '--cblk <actual block number>' parameter within the **blockserver.py** script. As a consequence, the server will output **"CORRUPTED_BLOCK <BLOCK_NUMBER>"**, and the client will indicate "**CORRUPTED_BLOCK <VIRTUAL_BLOCK_NUMBER>"** if attempts are made to access the degraded block.

For simulating server disks failing, first save data across several disseminated blocks and subsequently terminate the server (excluding the one hosting the RSM), using the **'Ctrl + C'** key combination. Assess the fault tolerance regarding the defective server by employing Unix shell commands. Upon encountering a faulty server, the system will exhibit **"SERVER_DISCONNECT <OPERATION> <VIRTUAL_BLOCK_NUMBER>"** alongside **"SERVER_DISCONNECT <SERVER_ID>"** to identify the unreachable server and the executed task. Alternatively, if the malfunctioning server's disk is substituted and reintegrated into the distributed file system, leverage the **'repair'** command accompanied by the flawed server's address. To authenticate successful data recovery, reapply the **showload** function preceding and succeeding the 'repair' function, signaling that requests have been forwarded to the problematic server, consequently restoring data using the distributed parity.

*Note:-Should any command lead to unexpected server closure, restart the servers and client adhering to the initial guidelines provided earlier. Before generating files and adding data, ensure that the requisite clients establish connections. Please note that there exists an outstanding concern within the codebase: initializing a fresh client after conducting certain file operations on the former client may result in modifications to the contents of the root directory's inode. This occurs due to change in the data blocks that associate with the root inode.*