Principles of Computer Systems Design - Final Project

Divy Nidhi Chhibber
MS, ECE, University of Florida

Murchhana Islam
MS, ECE, University of Florida

***Abstract*—This report is on the work done towards developing a client/server based file system, with multiple servers supporting redundant block storage. The system stores data across multiple servers on the block level to facilitate load reduction and replication of data by mirroring, increased capacity and fault tolerance. The design can tolerate a fail-stop failure of either the principal server or the mirror server in each mirror set, similar to the approach in RAID-1. The server side contains all of the data blocks. Two important functionalities, write and read, are implemented. The write function writes data to the data blocks on the server side and maintains redundancy by writing to multiple servers at once and storing copies in replicas. The read function reads the data blocks from the server side and maintains fault tolerance by reading from one replica even when its principal server is down or vice-versa. The read function is also load balanced as it reads from different servers in a mirrored set each time the function is called. The other functions implemented are those which are required for making directories, creating files, moving files, removing files, checking the status of the local file system or any of the server file systems, and finally exiting the program, all from the terminal command line by the user. Python 3+ has been used in all of the programming.**

***Index Terms*—Client/Server File System, Block Storage, Redundancy, Mirroring, Fault Tolerance**

I. Introduction - A Brief Problem Statement

The client-server architecture revolves around distributed computing, forming the basic idea of the modern network computing. The services are requested by the client, and the servers provide the resources for the service. The client requests the contents of a server, but it does not share any of its own resources. The client-server model may be on the same computer, or the client may initiate contact with the server which is on separate hardware over a network. This model forms the basis for Email and the Internet's TCP/IP.

In general, the client is a process which requests a service or a function over a computer network, and the server is a process which provides it. In our implementation, multiple servers are used to provide one user-operated client with any services or information requested. The services include making a directory, creating a file, writing data to the file, reading data from the file, moving and removing files, viewing the status of the file system and other evaluation metrics to check the client-server model's performance.

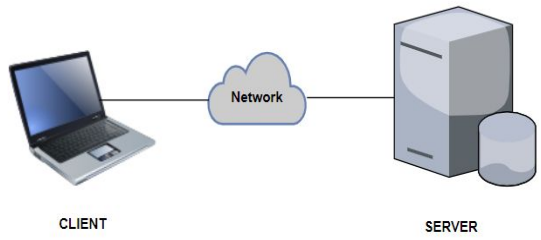

Fig. 1: General Client-Server Architecture

The client requires the location of the servers prior to establishing any connection with them. The locations are given by the server port numbers, which are the endpoints of network communication for the processes. The port numbers are provided by the user. Upon the establishment of connection, the client may request any service from the user interface (command line, in this case).

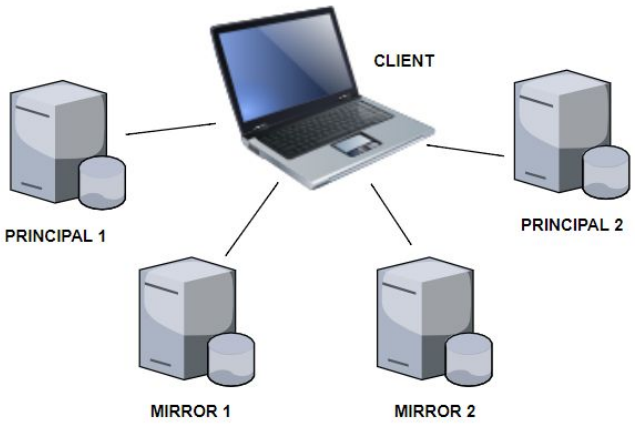

Fig. 2: Server Database Mirroring

One of the most important features of this client and

multi-server implementation is the use of database mirroring, which is used to create and maintain redundant copies of a database. This helps to avoid server downtime with continuous availability of uncorrupted data. This data redundancy is achieved by distributing the requests across (N, N+1) servers, where the (N+1) servers hold the mirror copies. Upon initialization of the system, a local file system and server file systems are created, along with both the principal and the mirror servers. When calling the write function, the data is written into the principal server blocks, with replicas simultaneously stored in the mirrors. So, in case of failure of a server, the data can still be recovered from the mirror during the read operations.

With the basics of the project introduced, this paper will now go on to discuss more about the design and the implementation of the system in Section II, the experiments conducted to evaluate the system in Sections III and IV, and finally conclude in Section V.

II. Design

A. Overview of the System Architecture and Main Modules

A basic schematic of the design of our client-system model with its modules is provided below.

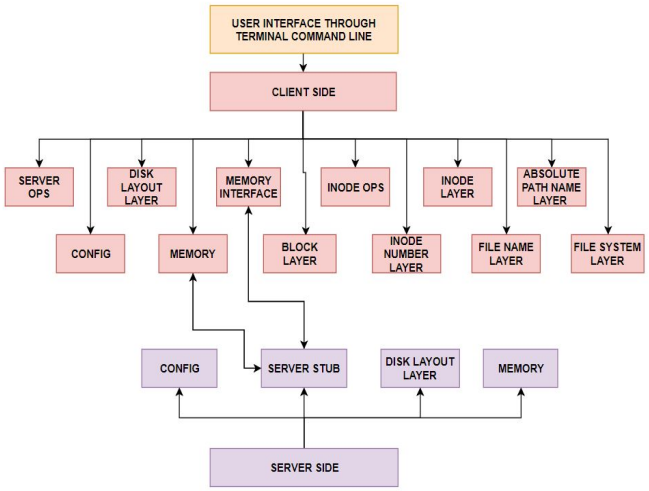

Fig. 3: Basic Schematic of the File System

Our file system has one client and multiple servers. The client takes in input from the user through the command line, where the user is asked by the client to enter the number of servers they want, the time delay they want for write/read acknowledgements, and the first port number of the first server. Every other server has its port number incremented by 1. For example, if the user chooses 4 servers to run, and puts the first port number as 8801, server 1 will have port 8802, server 2 will have port 8802, server 3 will have 8803 and server 4 will have 8804. The servers here are in an (N,N+1) mirrored set, same as in RAID-1 disk mirroring. For every odd numbered server (1, 3, 5, etc.), there is a replica in the even numbered server (2, 4, 6, etc.).

The client side stores a Local File System to store all the information related to the directories made and the files created. Apart from all of the inode blocks, bitmap blocks and superblocks, the entire inode table and its operations are stored here as well.

The server side holds the Server File System and is responsible for the data storage. The client's MemoryInterface is mapped to the server's ServerStub, with the help of other modules like ServerOps, InodeOps, etc. (which we have discussed about in detail in the following paragraphs) on the client side. The ServerOps module has a separate class for server initialization called Server_Init(), which has the attributes of the server proxy, name, number, time modified, created, accessed, and server state. In the InodeOps module, the structure of the Table_Inode() has been modified to include a self.ser_numbers, just as the self.blk_numbers already provided.

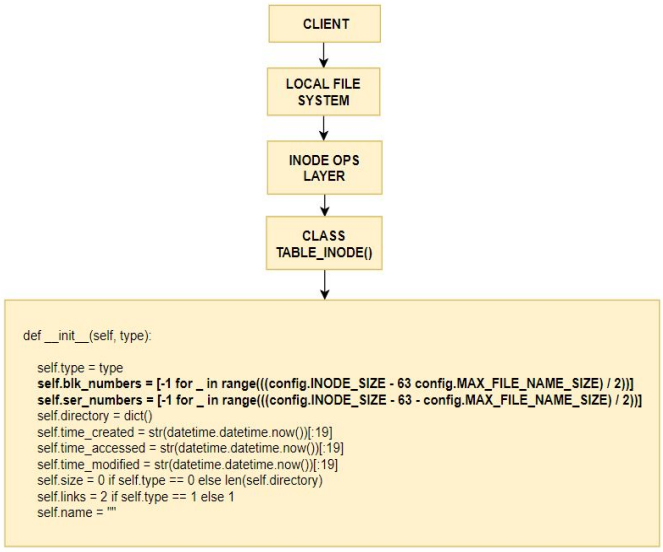

Fig. 4: Table_Inode() Class in InodeOps Module

This is the main design of the server and the data block storage. The Server File System thus makes sure that with each inode is associated a server, and its corresponding data blocks, to which data is written to and read from the client side over the network.

The client end has 12 modules. They are ServerOps, config, DiskLayout, Memory, MemoryInterface, BlockLayer, InodeOps, InodeNumberLayer, InodeLayer, FileNameLayer, AbsolutePathNameLayer and FileSystem. They are described as follows:

- ServerOps - This module is used for server initialization with the server proxy and the state.
- config - This module sets the values for the number

of blocks, the block size, the maximum number of inodes, the inode size, and the maximum file name size.

- DiskLayout - DiskLayout has definitions of the superblock and its attributes, and the bitmap blocks, the inode blocks and the data blocks.

- Memory - As the name suggests, acts like the memory of the file system with all the file system operations in it. The module includes a pointer which points to the DiskLayout module. It functions very similarly to the standard UNIX/LINUX file system.

- MemoryInterface - This is the core of the client-server model. It initializes the file system, boots all the servers for reading, writing and receiving acknowledgements upon completion of an action, and checking the server status. The functions for all these services are written here and the data is marshaled/pickled and sent across the network.

- BlockLayer - Updating data blocks, getting valid data blocks and freeing data blockings for read and write operations are done here.

- InodeOps - It contains all structures definitions and operations regarding converting the inode tables to arrays and arrays to tables.

- InodeNumberLayer - The module interacts with the MemoryInterface layer and updates the inode tables and inode numbers of the file system. Read, wrte, link and unlink operations across the server file system and the main file system are defined here.

- InodeLayer - This module provides the actual block number saved in the inode array of block numbers and fetched data/global handle of the block layer of the application programming interface.

- FileNameLayer - Sitting above the InodeLayer, the FileNameLayer acts as both the path and file name layers. For a particular directory, it uses a LookUp function to get the parent and child inode numbers associated with the path name and he file name, and updates them accordingly.

- AbsolutePathNameLayer - This sits atop the file system and interconnects all the file system operations. It provides the absolute path, that is, the path corresponding to the root directory.

- FileSystem - This initializes the main and the server file systems. Its main function starts all the functions and receives input for performing operations from the command line. This is the Python file that needs to be executed from the client side to get the file system up and running.

Similarly, the server side has 4 modules. They are config, DiskLayout, Memory and ServerStub. While the first three are same as that of the client side, the last one is a new module solely for the server.

- ServerStub - This initializes the server and establishes contact with the client across the network with its IP address/localhost and port number. This allows the client to remotely call procedures from the server with during an RPC. This is the server's main module responsible for the distributed computing aspect of our project.

The file system follows the approach RAID 1, which is also known as disk mirroring, where data is replicated into two or more disks. This is done mostly on the InodeLayer and the InodeNumberLayer, as the data is mirrored into a set of (N,N+1) servers as mentioned before. So, in the event that one server fails, the other server in the pair is present for the client to request its services. The data transferred across the network through read and write operations are marshaled within the Memory and MemoryInterface layers. The services are described towards the end of this section.

B. Implementation of Functionalities

The main functionalities implemented are status, mkdir, create, write, read, mv, rm and exit. These are the services which are requested by the client from the multiple servers through command line inputs. As mentioned in the previous section, the number of servers are first requested, along with a wait time and the first port number. The rest of the port numbers are in increments of 1.

The file systems are initialized, including the Main/Local File System and each individual Server File System, by calling two separate initialization functions from the MemoryInterface layer (Initialize_My_FileSystem() and Initialize_MFS()). A call is also made to the new_entry() and new_entry_MFS() functions in the AbsolutePathNameLayer to make the root directories as soon as the file systems are initialized.

In the MemoryInterface layer, the Initialize_MFS() function activates the state of the Main File System by calling the Initialize() function in the Memory layer. The Init() function here takes the number of servers and the first port number as its arguments. For each server that is activated, it creates and appends the server handler, creates the server proxy with a call to the Initialize_Servers() function, and then creates a server

instance with the Server_Init() function in the ServerOps layer mentioned before. It stores the server instance for a server in an ordered dictionary called dicton with its key, increases the port number by 1, and then continues to do the above in a loop for all the servers that have been asked to be initialized by the user. Initialize_Servers() uses XML-RPC over the localhost and the port number to make the server handler for the server proxy made above. Then, Initialize_My_FileSystem() initializes the individual Server File Systems with the help of server proxies made from the keys and the values stored in dicton. With the creation of the servers, blocks are also created on the server side with the InodeOps layer's operations. The file systems now have a root directory, and are now further ready to make and store new directories and files on the Local File System, and write to and read from data blocks associated with those files on the Server File Systems.

The user then begins implementing the functions from the command line. The most basic function that be implemented here is the status function. "$ status" is used to check the status of either the Local File System or the Server File Systems for every individual server that is active. The user is first asked which status he wants to see. For the Local File System, status_MFS() shows the status (same as was done in previous assignments). For the Server File Systems, status() first asks the user which server status they want to see. Using this provided server number, a server instance is created, whose server proxy is used to check that particular server's status.

The next function is the make directory or the mkdir function. This is solely for the Local File System, as all directories are stored on the client side. A directory is made according to the path specified. "$ mkdir A" will create directory A under the root directory in the main file system. The new_entry_MFS() functions in the AbsolutePathNameLayer and the FileNameLayer associate a new inode with the directory and assign a child inode number to the parent inode number, or create a completely new inode number for the special case of initializing the directory in the file system for the first time. The corresponding inode tables are also updated.

The create function also works in the same way for the Local File System for creating files that are also stored on the client side. A file is created under the path given in the file system. "$ A/1.txt" creates a text file called 1.txt under the directory A under the root directory with the help of the new_entry_MFS() functions in the AbsolutePathNameLayer and the FileNameLayer, by again associating inodes to it and updating the corresponding inode tables.

For the write function, upon entering "$ write", the system asks for the path where the data is to be written, the data that is to be written and the offset from where the data is written. The data provided is now stored across the data blocks of each server in a redundant manner by following a series a steps. First, the call to the write function from the FileSystem layer calls the write functions in the AbsolutePathNameLayer, FileNameLayer, InodeNumberLayer and finally the InodeLayer. The system first checks if the file exists or not in the Main File System, and if it exists, the InodeLayer write function is called. Here, the server numbers and block numbers of the inode are updated with each function call, with inode.ser_numbers and inode_blk_numbers. Then, the get_valid_data_block() function is called from the BlockLayer and the MemoryInterface layer. This is the function which gets the valid data blocks from the valid servers to which data will be written to. It first checks the server state and sends an acknowledgement if the server is alive and its state is 1. The least recently used server is first found with the help of the ordered dictionary, dicton, and the server proxy corresponding to it is used now. The data to be written is pickled /marshaled as a message and appended to the data blocks corresponding to the servers of the server proxies. Data redundancy is applied here, with the data being written to both the principal servers and the mirrored servers. With each data block having a size of 512, writing A*2018 times will store data in the first block of each server for four servers, along with the copies in the mirrors. Acknowledgements are sent from the servers prior to writing to let the user know to which servers data is being written to with the the No_Writing_Blocks() function in the MemoryInterface layer.

For the read function, "$ read" asks for the path of the data which is to be read, the offset from which it is to be read and the length of the data to be read. Because data redundancy has been used, even when one server from a mirror pair has failed, the other server can still fetch the data the user wants. Similar to write, the read function is called through several layers, and finally in the MemoryInterface layer, the Read_Data_Servers() function takes in the server numbers to which data have been written to and first decides the servers from which data is to be read (which is half that number - the principal servers or the mirrors). With dicton, the ordered servers are appended in a list and are ready to be read. Acknowledgements are also sent here with the Read_Server_Ack() function to let the user know from which servers data is being read. The get_data_block() function is used to fetch the data blocks from the servers and thus read is implemented.

The move ("$ mv") and the remove ("$ rm") function calls from the command line use the link() and unlink() functions, defined in the FileNameLayer and the InodeNumberLayer. For the move function, the old path and the new path have to be mentioned by the user, to specify which file they want to move and to where. $ mv /A/1.txt /B will move 1.txt from directory A by unlinking it and move it to directory B by linking it. The remove function is used to remove a file or a directory, so the path has to be provided. $ rm /A/1.txt will remove 1.txt from A by unlinking it. The link function works in the Main File System. It gets the inode number of the last file or directory in the paths, and creates a hard link. The inode tables are simultaneously updated and the reference count of

the number of links is incremented at the same time. The unlink function works in both the Main File System and the Server File System. For this, two different versions of update inode table and free data block functions are used, one for the Main File System, and one for the Server one. Thus, data written into the data blocks of the servers can be removed, along with any files and directories stored in the Local File System.

The final user function is the exit function, which is used to exit from the client-server file system model by the simple "$ exit".

C. Data Structures Implemented

The data structures implemented in the project are lists, sets and dictionaries.

- List - A list, written with square brackets, is a data structure in Python which is ordered and is also changeable. Thus, lists are different from other data structures like tuples, which are ordered but are not changeable, and sets, which are neither ordered, nor indexed. Several lists have been used in the modules because of the versatility of the collection.

  Lists have been used to access items and change their values. For example, adding a file name of the child with its inode number to the parent inode directory in the link() function in the InodeNumberLayer module. Lists have been looped through to either print their contents or to check whether a certain item exists in the list or not, as has been done in several of the functions comparing with a value in the list or along its length. Special list methods like append(), sort(), del(), *etc.* are also used. For instance, to append the server handler used in the Init() function in the MemoryInterface layer, we have the server list. To delete the file with the child name from the parent directory in the unlink() function in the InodeNumberLayer, we use the del statement.

- Set - As mentioned earlier, a set is a data structure in Python which is unindexed and unordered. Sets are written with curly braces. Several sets are used in the modules to access their values, update their values, finding their lengths, *etc.* For example, in the Read_Data_Servers() function in the MemoryInterface layer, the set ser_num is used to update its values according to the number of servers from which data is to be read, sorted depending upon which server was last written to (This has to be done using a set as the server mirroring approach would fix the server order when using a list. We need to use a set on the off chance that one of the (N,N+1) mirrored servers might fail, but the data still must be read.), and then looped through to append the values to a separate ordered read_servers list.

- Dictionary - This special Python data structure is changeable, indexed and unordered. Written with curly braces, dictionaries have keys and values. The values are accessed by reference to the keys using square brackets. In the MemoryInterface module, a dictionary is made from the imported collections module, which contains high performance data types, including OrderedDict(). Ordered dictionaries are similar to normal dictionaries in Python, they only additionally keep a note of the order in which items were added to them. In the Initialize_MFS() function, a dicton is made with the OrderedDict() method and values of server instances are appended to it through a loop over the number of servers initialized by the client. The dicton.get() method is used several times in the script thereafter to access the server instances required by the various functions in the MemoryInterface module.

III. TESTING

To test our implementation, we have performed several experiments including, but not limited to, the following:

- waiting for the all the servers that the user has requested from the command line to become alive so that the further operations can be performed properly
- acknowledging establishment of connection from the servers prior to initializing the file system so that if any server is not alive, the system will wait for it and inform the user
- making directories with mkdir /A, mkdir /B, *etc.* in the main file system
- creating files with create /A/1.txt in the main file system
- throwing an error when trying to read or write from a non-existent file or directory
- throwing an error when trying to write to a directory as only files can be written to
- throwing an error when trying to write from an incorrect offset
- sending acknowledgements of which servers are alive prior to writing so that the user is aware and can keep track of these servers

- writing to a file with write /A/1.txt, a*2048, 0 (data distributed across 4 data blocks across 4 servers including mirrored servers with data redundancy)
- writing data to one replica in case the the other replica is down (described in detail below)
- throwing an error when trying to read data from a directory as only data in files can be read from
- sending acknowledgements of which servers are alive prior to read operation so that the user knows where to read from and which servers are alive
- checking for fault tolerance for when one server is down and data is read from the alive replica with read /A/1.txt, 0, 2048 (described in detail below)
- checking for balanced reads (described in detail below)
- moving file from one directory to another
- throwing an error when trying to move the root directory into another directory
- moving one directory into another directory
- throwing an error when trying to move a parent directory into a child directory
- throwing an error when trying to remove a wrong or non-existent path
- throwing an error when trying to remove a non-empty directory
- removing a file
- removing an empty directory
- throwing an error when trying to remove the root directory
- checking the status of the local file system
- checking the status of the server file systems
- exiting from the program

For the write function tests, we first call the function. The servers then wait for the duration of the provided time delay and send acknowledgements of which servers are alive. They again wait for the delay time and then begin to write to the servers (principal and replica included) one by one. During this time delay, any one of the servers from a mirrored set can be turned off/stopped. The system then sends the acknowledgement from only the server of the mirrored pair that is alive and data is only written to that server.

Similarly, for the read function, the system first displays the servers on which the data resides. These can be either an even number of servers, or an odd number of server sif one of them had been turned off during the write operation. The system then waits for the time delay and sends acknowledgements from the servers that are alive, again waits, and then reads data from the servers. The fault tolerance level or order of the system is 1, that is, it can handle one server of a mirrored pair being down. So, if one server is down, it is tested to ensure that the data is read from its replica or mirror. If both servers are up, then the read is balanced. The system is tested to see whether for the first read the servers which fetch the data are different from the servers which fetch the data during the second read.

All these tests are performed to ensure that the project has been implemented correctly and that the read and write functionalities work according to what was asked for. The repeated tests confirm that the client-multiple server system works perfectly well data redundancy and fault tolerance.

IV. Evaluation and Extra Credit Work

To evaluate the performance of our implementation, we looked at the total time performance of the read and write operations of our file system, along with the previously implemented local memory file system and client/single server file system.

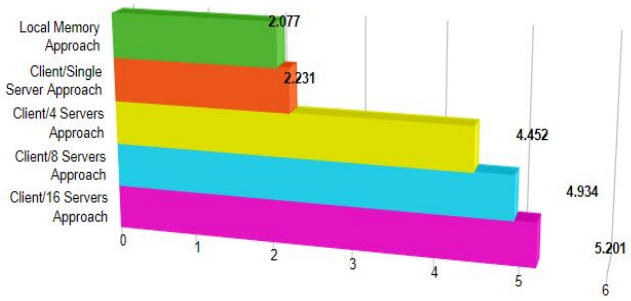

Fig. 5: Histogram Showing Total Time Performance of Execution across all File Systems

For the local memory and single server systems, the performance is quite fast. In the local memory approach, no connection has to be established across a network so the operations are performed just over two seconds. For the client/single server approach, with only one server is used, the performance is still very fast. However, for the client/multiple server approach, the performance is a lot slower with increasing number of servers as more connections are to be

established, with more acknowledgements and a much greater number of data blocks to be accessed to perform every read and write operation.

Table 1: Overall Performance Times of the File System Models

File System Architecture	Overall Performance Time
Local Memory Approach	2.077 seconds
Client/Single Server Approach	2.231 seconds
Client/4 Servers Approach	4.452 seconds
Client/8 Servers Approach	4.934 seconds
Client/16 Servers Approach	5.201 seconds

We also measured the operational performances of the read and write functions individually for each file system. Again, the local memory and single server approaches are still faster than the multiple server approaches. However, though the multiple server architecture requires a bit more time, it is much more fault tolerant than the other two architectures. Even when one server from one mirror set is down, data can still be written into and fetched from the file system because of its distributed computing architecture.

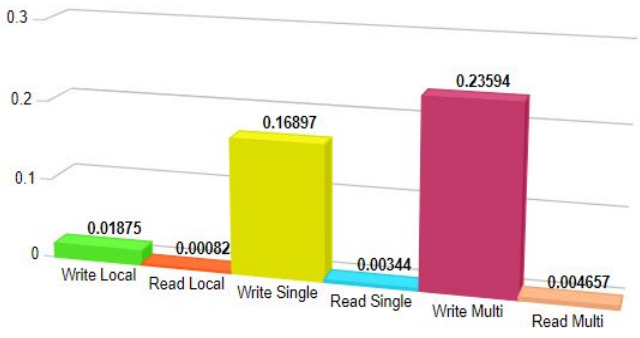

Fig. 6: Histogram Showing Time Performance of Individual Read and Write Functionalities across all File Systems

Table 2: Comparison of the File System Models

File System Architec-ture	Time Perfor-mance	Overall Performance	Fault Tolerance, Data Redun-dancy
Local Memory Approach	Comparati vely faster	Does not use RPC. Any error in data will corrupt whole system.	No.
Client/Singl e Server Approach	Comparati vely faster	Uses RPC. Error in server does not cause client to fail, however data will no longer be sent or received.	No.
Client Multiple Server Approach	Comparati vely slower	Uses RPC and mirroring. Error in one server will not cause the client to fail. Additionally, its mirror server will still be able to function with write and read.	Yes. Writes and reads are balanced across all the servers.

To compare the write latency in this project, with and without failure, we first write to the data blocks when four servers are on, and then see what happens when we turn down a mirror server and try to write.

Table 3: Write Latency for Failure and without Failure

Server	Write Time
All 4 servers on	0.172269 seconds
A server turned down	0.10530 seconds

For the extra credit part, we have used Amazon Web Services (AWS), a subsidiary of Amazon which provides Cloud Computing Services. Amazon Elastic Compute Cloud (EC2) has been used for this purpose. A server instance has been created, specifying its type (t2.micro) and its AMI ID (with Ubuntu 16.04).

The changes made to our code are in the MemoryInterface layer, where the Initialize_Servers() function has its localhost changed to the public IP address 18.222.205.249, provided to us by AWS. This generates the appropriate server proxy and handler required to initialize the servers on the cloud instance.

For this purpose, terminal emulator PuTTY and FTP and SCP client WinSCP are used to make sure that the client on our system communicates with the servers on the AWS, and securely transfers the files from the local to the remote systems. PuTTYgen converts the private key downloaded from EC2 to create an SSH key for PuTTY to use. WinSCP transfers the server files to the server instance on the cloud. Hostnames are provided on the PuTTY terminal emulator and the desired number of servers can be entered in by the user and the operations can begin. Each PuTTY terminal window executes different ServerStub modules for each server that has

been initialized by the user. With all the servers running, the functions are all executed one by one by the user. The protocol used here is TCP. The cloud here acts as the memory of the system, with the data getting stored here.

To check the cloud server instance performance, we first looked at CloudWatch. Amazon CloudWatch is a cloud monitoring service provided by AWS to collect data, monitor log, metrics, etc.

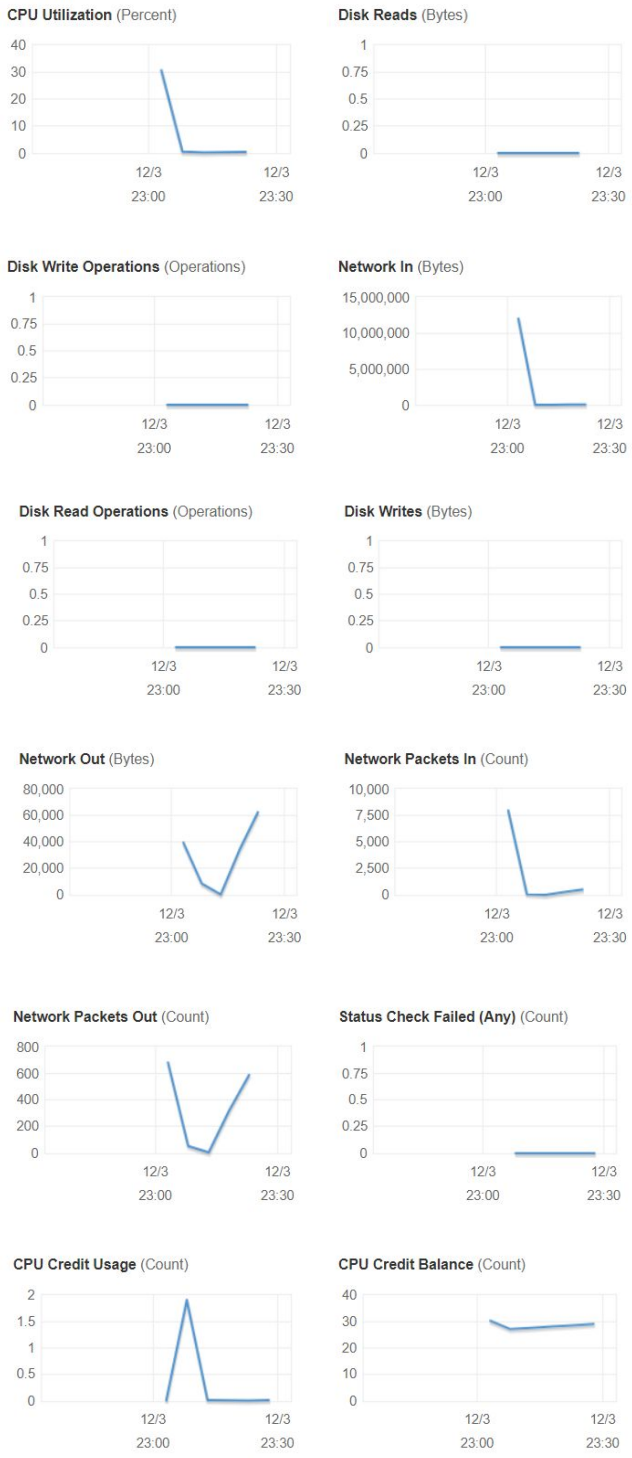

Fig. 7: Amazon Web Services CloudWatch Evaluation Metrics

Secondly, we compared it to our previous file systems to check for its read and write function performances.

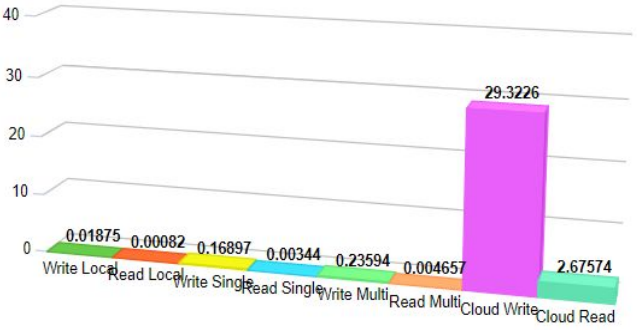

Fig. 8: Histogram Showing Time Performance of Individual Read and Write Functionalities across all File Systems including the Cloud Instance

We see that the time required for the functions, especially the write function, are extremely higher than the times for the write functions of all the other file systems. This is because the time taken for the connections to be made over the TCP network are a lot more than for a client-service architecture on the same computer, or the local memory approach.

For different number of servers activated on the cloud, we observe that the time keeps increasing to perform the operations as the number of servers also increases.

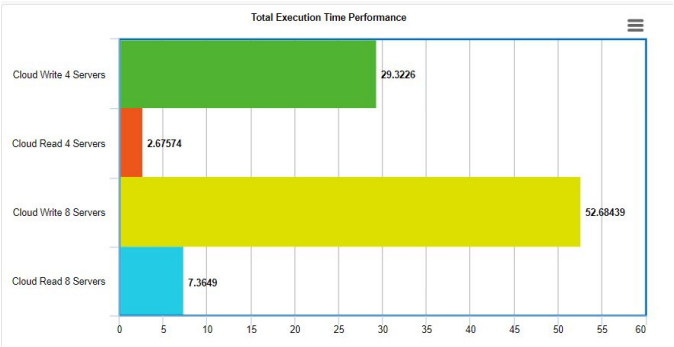

Fig. 9: Histogram Showing Time Performance of Individual Read and Write Functionalities for 4 and 8 Servers on the Cloud Instance

Thus, though the execution time is a lot more for the functions, the advantages of the cloud instance is a lot more than the client-service model on the same computer.

The advantages of having the server instances on the cloud including:

- Scalable - Change in size according to the amount of data that is required to be stored
- Centralized and Accessible from Anywhere - The biggest advantage that cloud computing offers us is that the data can be accessed anytime, anywhere and anyone can contribute to it
- Security - More secure than a standard computer server with high-level network encryption
- Cost-effective - Saves money from buying expensive hardware in the long run

V. Conclusions

We see that our implementation of the project checks all the boxes which were required to be designed and analyzed. The data servers have been mirrored, similar to RAID-1. The servers store the data blocks, and the information of the corresponding data blocks and server numbers are stored in the inode table on the client side. The client side also has a Local File System, which keeps track of most functions and the inodes of all directories and files. The server side has a Server File System for each individual server.

Data is written into the servers by maintaining redundancy and mirroring the data into a server replica. Data is read from the servers by maintaining fault tolerance of the order of 1 so that when a server is down, its replica can fetch the data from its data blocks for the read operation. The read function is also load balanced, so that when one read call reads from a particular set of servers, the next read call reads from their mirrors.

The performance evaluation of all the file systems and the extra credit analysis has also been done. The servers run properly on the cloud instance through Amazon Web Services.

VI. References

[1] Deepali Mittal, Neha Agarwal, "A review paper on Fault Tolerance in Cloud Computing," *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), 11-13 March 2015*

[2] Shenze Chen, D. Towsley, "A performance evaluation of RAID architectures," IEEE Transactions on Computers (Volume: 45 , Issue: 10 , Oct 1996)

[3] H.H. Kari, K. Saikkonen, Sungsoo Kim, F. Lombardi, "Repair algorithms for mirrored disk systems," Proceedings of International Workshop on Defect and Fault Tolerance in VLSI, Nov 1995

[4] Panmin Huang ; Rongliang Wang, "Client-server upgrade model of design and realization based on embedded linux," *2011 3rd International Conference on Computer Research and Development, March 2011*

VII. Division of Labor

The project has been done in a group of two:

Divy Nidhi Chhibber - Designing the file system architecture, writing code for the same, performing tests, analyzing performance of systems, writing the report.

Murchhana Islam - Writing code, performing tests, performance, writing the report.