GROUP C1

Ekal Golas, Jayakarthigayan Sridharan, Ketan Joshi, Sahith Katukuri

UNIVERSITY OF TEXAS AT DALLAS | 800 W CaMPBELL RD RICHARDSON TX 75080

PROJECT REPORT

CLOUD COMPUTING PROJECT



Table of Contents

[Group Information 3](#_Toc436500440)

[Details of team members 3](#_Toc436500441)

[Introduction 4](#_Toc436500442)

[Goal of the project 4](#_Toc436500443)

[Functionalities provided 4](#_Toc436500444)

[Study of related work 8](#_Toc436500445)

[Summary of related works 8](#_Toc436500446)

[Comparison with existing works 8](#_Toc436500447)

[System architecture 10](#_Toc436500448)

[Activity Diagram 10](#_Toc436500449)

[Architecture Diagrams 11](#_Toc436500450)

[Detailed Design 15](#_Toc436500451)

[Client 15](#_Toc436500452)

[Commons 16](#_Toc436500453)

[Master 16](#_Toc436500454)

[GFS Directory Operations 17](#_Toc436500455)

[Ceph Directory Operations 19](#_Toc436500456)

[NFS Directory Operations 25](#_Toc436500457)

[Implementation Details 28](#_Toc436500458)

[Client 28](#_Toc436500459)

[Commons 28](#_Toc436500460)

[Master 30](#_Toc436500461)

[Problems Encountered 33](#_Toc436500462)

[Experimental Results 35](#_Toc436500463)

[Performance by each solution 35](#_Toc436500464)

[Performance by each command type 35](#_Toc436500465)

[Performance by node depth 35](#_Toc436500466)

[Installation Manual 36](#_Toc436500467)

[User Manual 38](#_Toc436500468)

[Workload Distribution 40](#_Toc436500469)

[Summary 40](#_Toc436500470)

[Detailed list 41](#_Toc436500471)

[References 47](#_Toc436500472)

# Group Information

Group: C

Project Name: Directory Structure Maintenance

Team Members: 4

TA: Yongtao Huang

## Details of team members

1. Ekal Golas – exg140230
2. Jayakarthigayan Sridharan – jxs143730
3. Ketan Joshi – kkj140030
4. Sahith Katukuri – sxk145130

# Introduction

## Goal of the project

The project problem statement (Yen, Cloud Computing, 2015) is stated as:-

Compare different methods in implementing directory files

* In Unix system, each directory is a file by itself
* In many distributed file system, files are treated as individual elements during placement among the distributed servers. Directory structure is handled separately.
  + Solution 1: Use a centralized server to store the entire directory
  + Solution 2: Treat directory files as regular files
  + Solution 3: Ceph solution
  + Solution 4: Merge a subtree of directories into one file, with a fix number of levels

Project steps

* Develop the four directory structure maintenance solutions and compare their performance
* Consider directory creation, deletion, rename, cd, ls operations
* Generate a huge directory structure
* Develop a directory request generation program to generate requests and evaluate the performance of different solutions

## Functionalities provided

#### Client

Running of the client should not be dependent on the master. This enables isolation and simplicity in the design

1. Configuration file to easily change client parameters – A configuration file will be provided with the client that will contain the following parameters:-
   1. Master IP and port – Information about the socket to connect to
   2. Input file path – File to parse directory structure from and output the result to a configurable file path which will contain a configurable number of weighted random commands
2. Command generation in a weighted random distribution – The client implements Zipf distribution to generate a configurable number of random commands that can be used to generate uneven load on different paths in the directory structure
3. A cache to store the MDS server information to which commands will be sent
4. Logging to track all requests and responses

#### Master

Similar to the client, running of the master is also not dependent on the client. A single master hosts three sockets for GFS, Ceph and NFS solution.

1. Configuration file to easily change client parameters – A configuration file will be provided with the master that will contain the following parameters:-
   1. Ports to host each directory solution
   2. Ceph configuration – Configuration for each MDS and replicas and their partition information
   3. Input file path – File to parse directory structure from
2. Multithreading – Each directory solution operates in its own thread and is unaffected by the operation of other threads
3. Synchronization - Concurrent reads and exclusive writes policy to access the resources in all directory solutions
4. Ability to serve multiple clients – For each solution, the hosted socket can serve multiple client requests at the same time by implementing concurrent reads and exclusive write capability
5. Performance logging – Logging of time taken by each command executed by the master

#### Commands Supported

Following commands will be supported for each solution:-

1. **LS <path>** - List all the subdirectories and files inside the directory (just display the names)
2. **LSL <path>** - Working similar to "ls -l"
3. **RMDIRF <path>** - Remove given directory (The option works like "rm -r -f" command. So the command will remove the directory even though it is not empty)
4. **RMDIR <path>** - Remove given directory (Fail and prompt user if the directory is not empty)
5. **MKDIR <path>** - Create mentioned directory (Fail if already present)
6. **TOUCH <filepath>** - Create given file in the mentioned directory. If already present, modify its timestamp
7. **CD <path>** - Change current working directory
8. **PWD** - Print current working directory

#### Directory Metadata

Following metadata information will be supported with each node in the directory structure:-

1. **Name** – Name of the file or directory
2. **isFile** – Flag to indicate if the node is a file or a directory
3. **Children** – List of files and directory in this directory
4. **Modified time stamp** – Time when the node was last modified
5. **Size** – Size of the file
6. **Inode** – Inode information including MDS details for Ceph solution
7. **Operation counter** – To track number of hits on this node

# Study of related work

## Summary of related works

Following sources were referred before implementing the design for this project:-

1. (Google, 2003) – Paper for GFS, to refer for implementation of the Google File System solution
2. (University of California, 2006) - Paper for Ceph, to refer for implementation of the Ceph File System solution
3. (Sun Microsystems, 1984) - Paper for NFS, to refer for implementation of the Network File System solution

## Comparison with existing works

#### Ceph

Comparing with other works, this project implements some details in Ceph differently as follows:-

1. Only the fields inode number, modified time, lock, name and size are supported in the metadata
2. The metadata storage solution implemented in this project follows a fine grained approach with dynamic partitioning
3. Logging to journals is not implemented, but the command output is written to a log file
4. MDS load balancing is not implemented, but the functionality to support it like operation counters is provided

#### GFS

Comparing with other works, this project implements some details in GFS differently as follows:-

1. This project does not implement obtaining the chunk assignment information from the chunk server
2. Existing works use prefix compression in the B-tree representation of the metadata, which is not implemented in this project
3. Creating checkpoints to store metadata on disk is not implemented in this project

#### NFS

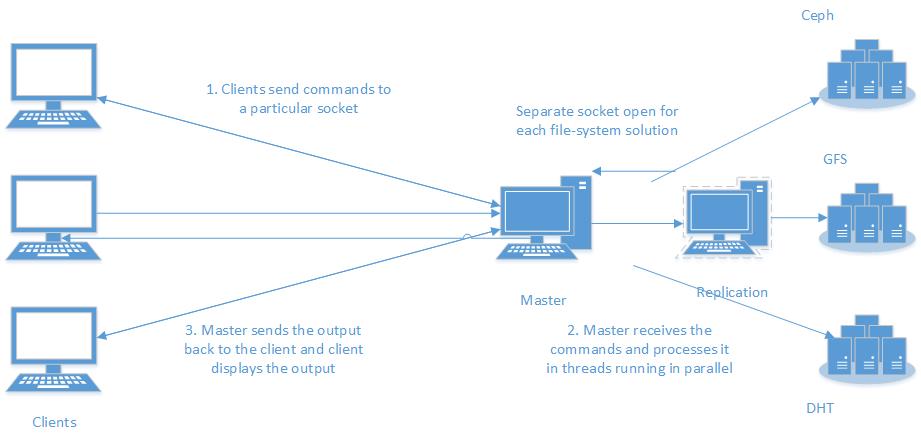
This solution was implemented with the design of DHT file system. As DHT file system was not obtained, NFS was used to place the files referenced by the metadata. Comparing the existing works for DHT implementation, this projects differs as follows:-

1. Replication was assumed to be provided, hence this project does not implement replication
2. Java`s filechannel API is used to implemented locking for the cut level file which stores the metadata. Locking for a particular directory is not implemented

# System architecture

## Activity Diagram

The following diagram depicts the design of client and the master components and the communication between them:-

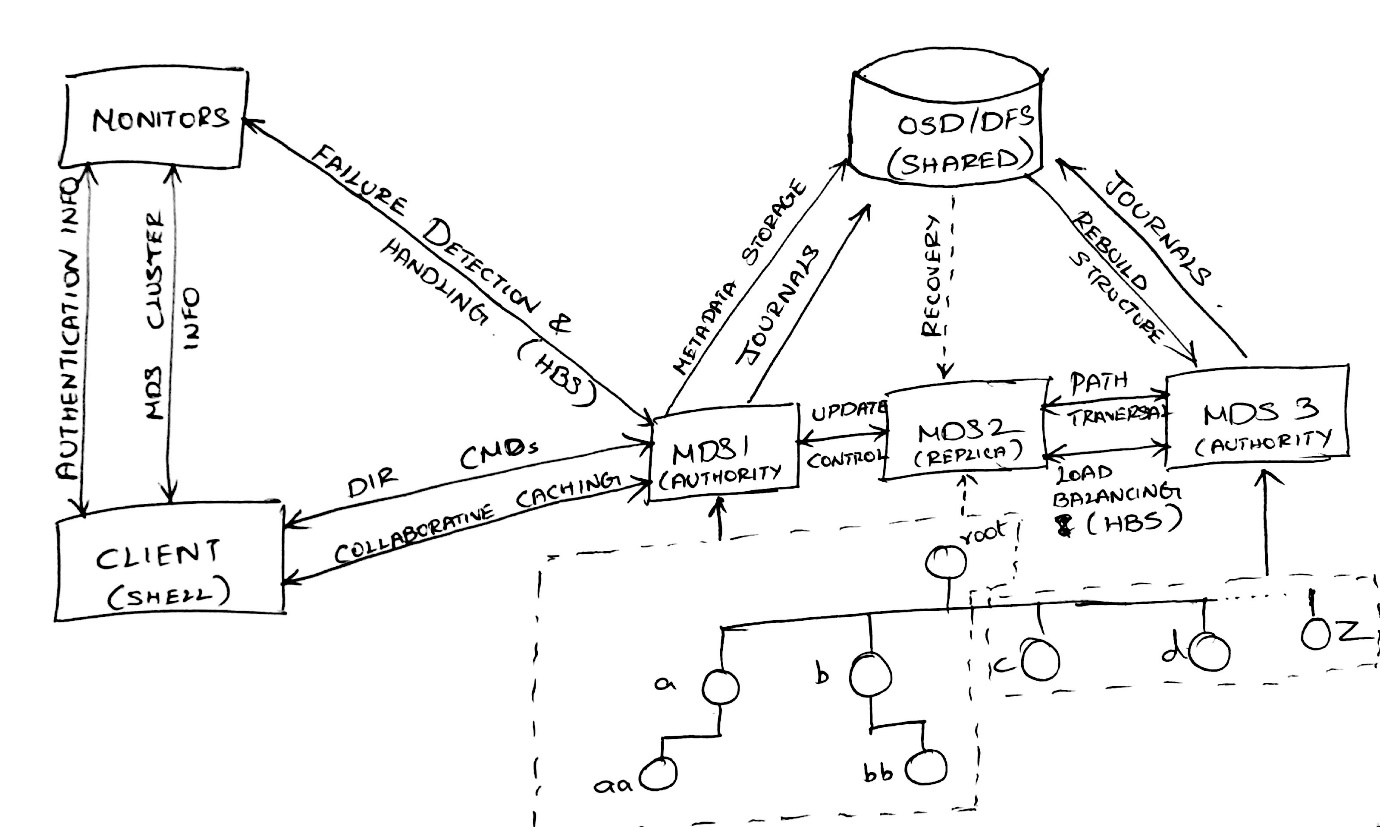


There are three components to the system, each represented as packages which represent the following:-

1. Client – All classes and packages inside it represents the client functionality and implementation
   1. Command generation
   2. Cache
   3. Communication with the master by socket programming
2. Master – All classes and packages inside it represents the master`s functionality, dependencies and implementation
   1. Ceph solution
   2. GFS solution
   3. NFS solution
   4. Communication with multiple clients with socket programming and multithreading
   5. Replication
3. Commons – A package required by both client and master to implemented common utility methods and class representations
   1. Directory representation
   2. Configuration and other utilities

## Architecture Diagrams

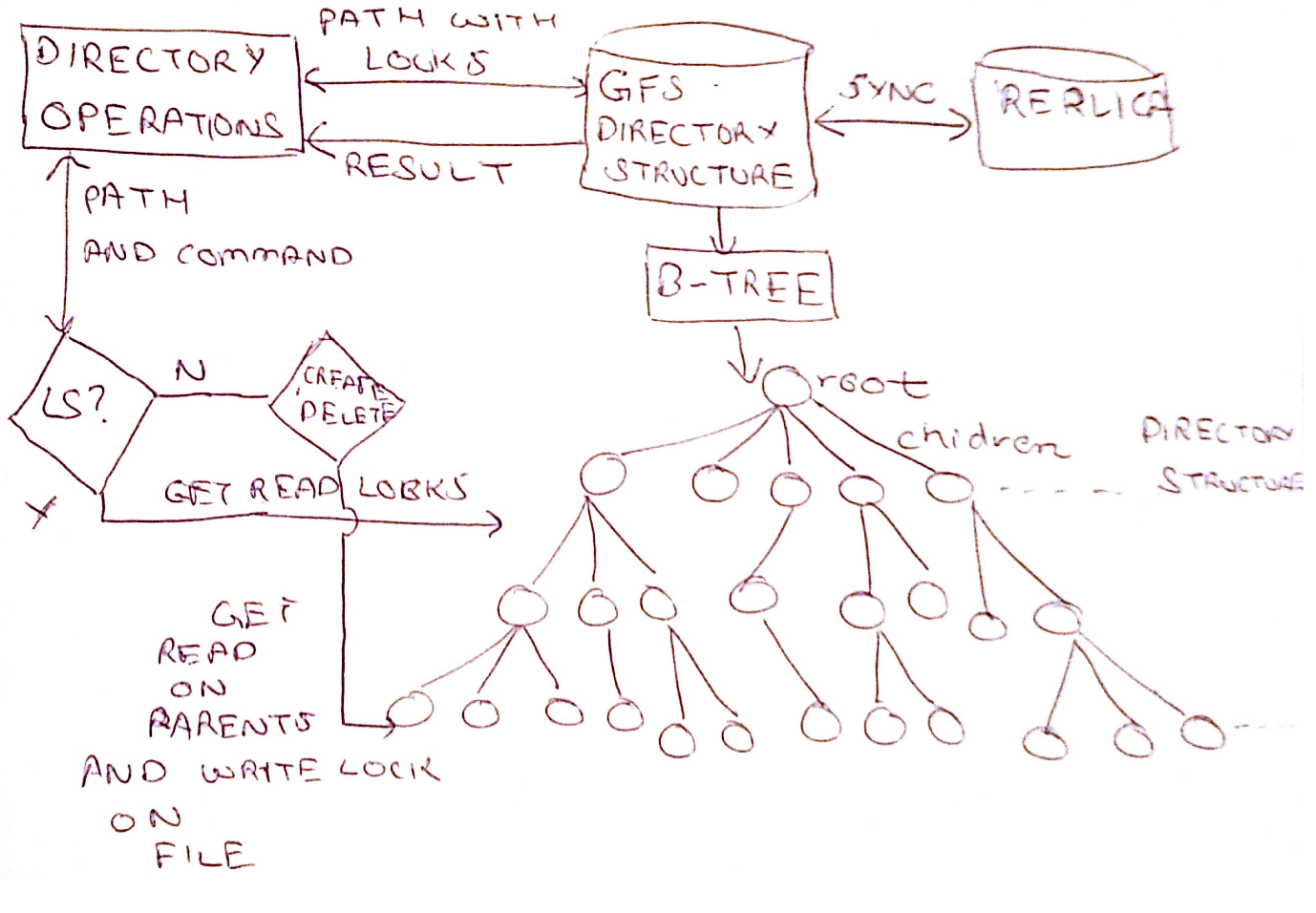
#### Ceph



The above diagram is explained as follows:-

1. Client tries to get the closest MDS server by collaborative caching
2. After the MDS server information is obtained, the client fetched locks on the resources
3. If it is a write operation, all the parent directories are read locked and the node is write locked
4. If it is a read operation, all the resources are read locked.
5. Multiple reads at the same time are allowed, but only single write can happen at any time. Also if a write happens, read has to wait and vice-versa
6. The command is then passed to the closest MDS server which then executes the command
7. The command is then executed on the primary MDS and the replica synchronously and the results are returned to the client. If the command fails, error message is returned to the client.

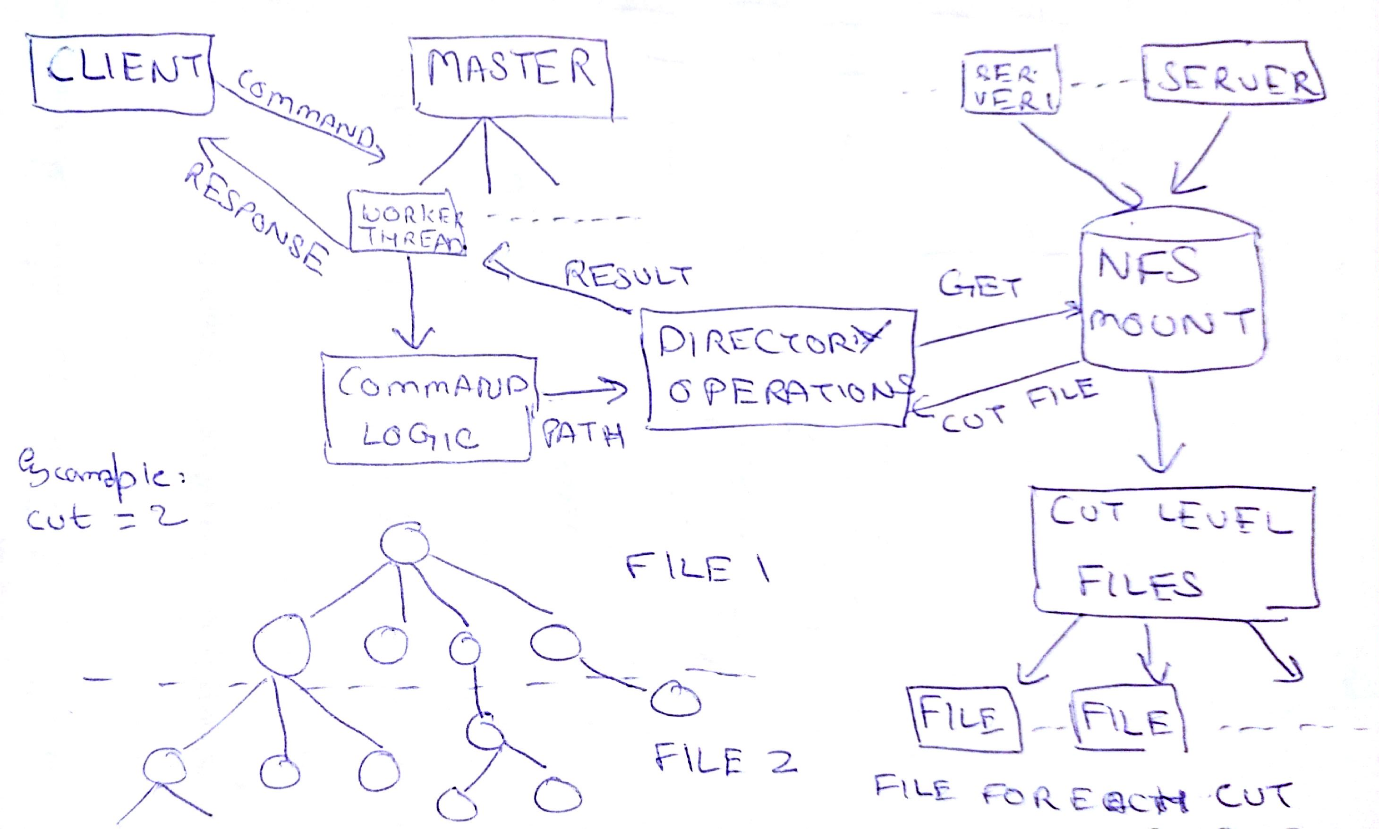
#### GFS



The above design is explained as follows:-

1. Master server gets the command from the client and calls Directory Operations
2. Directory Operations then gets the path of the directory and tries to acquire locks for concurrent access
3. If it is a write operation, all the parent directories are read locked and the node is write locked
4. If it is a read operation, all the resources are read locked.
5. Multiple reads at the same time are allowed, but only single write can happen at any time. Also if a write happens, read has to wait and vice-versa
6. The command is then executed on the primary and the replica synchronously and the results are returned to the client. If the command fails, error message is returned to the client.

#### NFS



The above design is explained as follows:-

1. Master server gets the command from the client and calls Directory Operations
2. Directory Operations then gets the path of the directory and tries to acquire locks for concurrent access
3. If it is a write operation, the cut file containing the directory is write locked
4. If it is a read operation, the cut file containing the directory is read locked.
5. Multiple reads at the same time are allowed, but only single write can happen at any time. Also if a write happens, read has to wait and vice-versa
6. The directory structure is written into cut files, where each cut is the height level on which the B-tree directory structure is cut and stored in a separate file
7. After the locks are acquired the command is then executed and the cut file is modified accordingly and written back to NFS and the results are returned to the client. If the command fails, error message is returned to the client.

# Detailed Design

## Client

The client is launched by calling the Client class. It defines a main method which carries out the client operations. After the client class is called and the client is started, the process flow is implemented as:-

1. As the client starts, it calls the AppConfig class in [commons](#_Commons) to load configuration
2. Gets the master`s IP address and port to communicate, and creates a socket on that
3. Uses CommandGenerator class to generate commands to the server.
   1. CommandGenerator class gets the directory structure from the file in the configuration
   2. Gets all possible paths in the directory structure as an array
   3. Randomly shuffles the list of command operations defined in CommandsSupported enum in [commons](#_Commons) and the array of all possible paths
   4. Uses Zipf distribution provided by Apache`s math library to get a weighted distribution of a preconfigured number of commands that will be sent to the server
4. Client obtains the commands and serializes the data using the Message class in [commons](#_Commons)
5. executes them sequentially and displays the output to the console using the OutputFormatter class in [commons](#_Commons)

## Commons

This component is defines utilities and classes used by both client and master. It contains the following:-

1. The code provided by the TA to implement communication between client and server using wrapper over TCP
2. The Directory class that represents the directory structure as a tree.
3. DirectoryParser class that reads an input file from configuration which has output of directory hierarchy and creates a directory structure using Directory class
4. Interface ICommandOperations, that defines the directory operations supported
5. AppConfig class to parse and load the configuration
6. Message class to serialize data for communication between client and master
7. OutputFormatter class to pretty print the output on the console
8. Globals class to store global metadata
9. CommandsSupported enum to list possible commands

## Master

This component defines each directory solution and the working of master server in order to communicate with clients. The master is started by launching the Master class. It defines a main method which carries out the server operations. The process flow is as follows:-

1. As the master starts, it calls the AppConfig class in [commons](#_Commons) to load configuration
2. Opens up a socket for each directory solution
3. Creates the directory structure using DirectoryParser class in [commons](#_Commons) and stores this data in a serialized format to a file read by configuration
4. Starts Worker threads for each socket that listen to clients
   1. As a thread receives a command from a client, it establishes connection with that client
   2. Gets the command and executes it on the desired directory solution
   3. Computes the output and sends it back to the client
   4. Repeats the process until an exit command is issued by the client

Master contains the implementation of different directory solutions. Each of them define their own implementation of commands defined in [commons](#_Commons). These implementations are as follows:-

## GFS Directory Operations

The commands for GFS are implemented as follows:-

1. **LS** – Performs a search operation on the tree to get the path that matches the path in the directory tree.
   1. Obtains a read lock on the directory and all its parents
   2. If the path is not found or the directory is empty, an error message is returned, else we use the OutputFormatter class in [commons](#_Commons) to display the children name and type
   3. Release the read locks acquired
   4. Send the results back to the client
2. **LSL** – Same as LS, except that we use the OutputFormatter class to display all the metadata of the directory
3. **MKDIR** – Performs a search similar to LS to get the parent directory.
   1. Obtains read locks on all the parents, and write lock on the current directory
   2. If LS returns an error, error is returned, else a new child node is created in the parent node as a directory.
   3. Releases the write lock and the read locks acquired
   4. If the operation was successful, executes the same command on the replica directory structure as well, acquiring and releasing locks in the same manner
   5. Sends a success message to the client if both primary and replica executed successfully, else send error message
4. **TOUCH** – Similar to MKDIR except that the new child gets added as a file.
5. **RMDIR** - Performs a search similar to LS to get the parent directory.
   1. Obtains read locks on all the parents, and write lock on the current directory
   2. If LS returns an error, error is returned, else the node matching the directory is removed from the list of child nodes of the parent directory only if the directory is empty
   3. Releases the write lock and the read locks acquired
   4. If the operation was successful, executes the same command on the replica directory structure as well, acquiring and releasing locks in the same manner
   5. Sends a success message to the client if successful on both primary and replica, error if the directory was not empty or any of the primary or replica failed
6. **RMDIRF** – Similar to RMDIR but removes the directory even if the directory is not empty
7. **CD** – The client stores the root of the directory structure in its cache. As the CD command is issued, a search is performed for the path in the directory structure similar to LS. If the path exists, root is changed to that path. After the CD command is issued, all relative paths are prepended with the path of the root stored in the cache before the command is sent to the server. If the path is absolute, no modification is done.
8. **PWD** – Print the current value of the root stored in the client`s cache

In memory replication is implemented by imitating the changes on the replica metadata based on the behavior as a result of the command executed on the primary metadata.

## Ceph Directory Operations

The commands for Ceph are implemented as follows:-

1. **LS** – Lists all the files in the provided path if the path points to a directory, else will list the details of the file in the path. If the path doesn’t exists (i.e. the path is neither a directory nor a file), then an error message is returned to the client. The implementation details of this command is as follows:
   1. Client acquires read locks for parents and current directory through the primary MDS server and proceeds if it can acquire all locks, else it errors out
   2. Client finds the closest MDS corresponding to the path
   3. Client sends the “**ls <file path>**” command to the server closest to the file path available in its cache. Initially it will be the server containing the root.
   4. MDS checks whether the provided path or a part of the path is in its list of sub-directories. If the path is not found in its list of sub-directories then will return an error to the client.
   5. If the path is found in the list of sub-directories, the MDS will start searching for the complete path in its directory structure. If it is reaches a dead-end where it cannot proceed further then throws error message to the client.
   6. If the expected file/directory is found then return the list of details for that node and its children if exists.
   7. If it reaches a node which resides in another MDS, forward the command to one of the MDS in the list and wait for the response from that MDS. Once it gets the response from the MDS containing the file/directory then return the same to the client.
   8. Client then tries to unlock all the read locks acquired. If it is successful, the command is a success, else a failure completion code is logged
2. **LSL** – Same as LS, except that we use the OutputFormatter class to display all the metadata of the directory
3. **MKDIR** – Create a directory in the specified path. If the path already exists or the parent directory not found then return an error message to the client. The implementation details for this command is as follows:
   1. Client acquires read locks for parents and write lock for the current directory through the primary MDS server and proceeds if it can acquire all locks, else it errors out
   2. Client finds the closest MDS corresponding to the path
   3. Client sends the “**mkdir <file path>**” command to the server closest to the file path available in its cache. Initially it will be the server containing the root.
   4. MDS checks whether the provided path or a part of the path is in its list of sub-directories. If the path is not found in its list of sub-directories then will return an error to the client.
   5. If the path is found in the list of sub-directories, the MDS will start searching for the complete path in its directory structure. If it is reaches a dead-end where it cannot proceed further then throws error message to the client.
   6. If the expected parent directory is found then check whether it is the authority for that directory. If it is not the authority, then forward the command to the authority MDS.
   7. If it reaches a node which resides in another MDS, forward the command to the authority MDS in the list and wait for the response from that MDS. Once it gets the response from the forwarded MDS then return the same to the client.
   8. If the current MDS is the authority for the parent directory and the new directory does not exists in the parent directory, then create the new directory and forward the command to the replicas to update all copies. Once all the all replicas are updated, the authority MDS will send the response back to the client.
   9. Client then tries to unlock all the read and write locks acquired. If it is successful, the command is a success, else a failure completion code is logged
4. **TOUCH** – Similar to MKDIR except that the new child gets added as a file. If the file/directory exists then the timestamp of the node is updated to the latest timestamp. The implementation details for this command is as follows:
   1. Client acquires read locks for parents and write lock for the current directory through the primary MDS server and proceeds if it can acquire all locks, else it errors out
   2. Client finds the closest MDS corresponding to the path
   3. Client sends the “**touch <file path>**” command to the server closest to the file path available in its cache. Initially it will be the server containing the root.
   4. MDS checks whether the provided path or a part of the path is in its list of sub-directories. If the path is not found in its list of sub-directories then will return an error to the client.
   5. If the path is found in the list of sub-directories, the MDS will start searching for the complete path in its directory structure. If it is reaches a dead-end where it cannot proceed further then throws error message to the client.
   6. If the expected file/directory is found then check whether it is the authority for the specified file/directory. If it is not the authority, then forward the command to the authority MDS.
   7. If it reaches a node which resides in another MDS, forward the command to the authority MDS in the list and wait for the response from that MDS. Once it gets the response from the forwarded MDS then return the same to the client.
   8. If the current MDS is the authority for the file/directory and the file/directory already exists, then update the timestamp of the node and forward the command to the replicas to update all copies. Once all the all replicas are updated, the authority MDS will send the response back to the client.
   9. If the expected file does not exists but the current node is the authority for the parent node, then create the file and forward the command to the replicas to update all copies. Once all the replicas are updated, the authority MDS will send the response back to the client.
   10. Client then tries to unlock all the read and write locks acquired. If it is successful, the command is a success, else a failure completion code is logged
5. **RMDIR** – Removes the specified directory. If the directory to be removed is not found, then throw error message to the client. The implementation details for this command is as follows:
   1. Client acquires read locks for parents and write lock for the current directory through the primary MDS server and proceeds if it can acquire all locks, else it errors out
   2. Client finds the closest MDS corresponding to the path
   3. Client sends the “**rmdir <file path>**” command to the server closest to the file path available in its cache. Initially it will be the server containing the root.
   4. MDS checks whether the provided path or a part of the path is in its list of sub-directories. If the path is not found in its list of sub-directories then will return an error to the client.
   5. If the path is found in the list of sub-directories, the MDS will start searching for the complete path in its directory structure. If it is reaches a dead-end where it cannot proceed further then throws error message to the client.
   6. If the expected directory is found then check whether it is the authority for that directory. If it is not the authority, then forward the command to the authority MDS.
   7. If it reaches a node which resides in another MDS, forward the command to the authority MDS in the list and wait for the response from that MDS. Once it gets the response from the forwarded MDS then return the same to the client.
   8. If the current MDS is the authority for the directory and the deletion directory already exists, then check whether the directory is empty. If the directory is not empty then return an error message to client.
   9. If the current MDS is the authority for the directory and the deletion directory already exists and the directory is empty, remove the directory and update the parent directory. Also forward the command to all the replicas and once it gets the reply from all the replicas, it response back to the client.
   10. Client then tries to unlock all the read and write locks acquired. If it is successful, the command is a success, else a failure completion code is logged
6. **RMDIRF** – Similar to RMDIR but removes the directory even if the directory is not empty
7. **CD** – The client stores the root of the directory structure in its cache. As the CD command is issued, a search is performed for the path in the directory structure similar to LS. If the path exists, root is changed to that path. After the CD command is issued, all relative paths are prepended with the path of the root stored in the cache before the command is sent to the server. If the path is absolute, no modification is done.
8. **PWD** – Print the current value of the root stored in the client`s cache

For the commands stated above, the replication is implemented by using an OperationCounter field in Directory class, which is incremented for a node every time a request passes through that particular node.

## NFS Directory Operations

The commands for NFS are implemented as follows:-

1. **LS** – Lists the directory information.
   1. It takes the file path as an argument and splits the file path based on the level cut number.
   2. It then takes the trimmed path and searches in the Hash map for the corresponding file.
   3. If the file is found, it obtains a read lock on the file. Otherwise, an error message is returned
   4. It reads the file line by line to look for corresponding path information and returns the lines which matches the path.
   5. These lines are processed to the get the directory metadata information and the result is calculated and sent to the client. If the path is not a directory or the path is not valid, error message is returned to the client.
   6. After the command is executed, the read lock on the file is given up.
2. **LSL** – Same as LS, except that we use the OutputFormatter class to display all the metadata of the directory
3. **MKDIR** – Creates a directory
   1. Performs a search similar to LS to get the corresponding file and search for the line which contains the given path
   2. A read lock is obtained on the file and a temp file is created with write lock on it
   3. If found throw an exception that directory exists else write a line with the given path in the temp file .If it is at the level cut number create a file with the full path written to it.
   4. After the data is written, delete the cut level file and rename the temp file to the cut level file
   5. Unlock all the read and write locks obtained on the cut level files.
   6. Return the results to the client
4. **TOUCH** – Similar to MKDIR search for the path in the file. If path is found then rewrite the line with the updated time stamp else append the path in the file.
5. **RMDIRF** – Removes a directory even if it is not empty
   1. Performs a search similar to LS and gets the cut file
   2. A read lock is obtained on the file and a temp file is created with write lock on it
   3. Write all the lines in the cut level file to the temp file except the lines which matches the pathname and also remove the files whose subdirectories information is stored in a cut-level file by recursively calling the command
   4. After the data is written, delete the cut level file and rename the temp file to the cut level file
   5. Unlock all the read and write locks obtained on the cut level files.
   6. Return the results to the client, error if parent directory did not exist.
6. **RMDIR** – Similar to RMDIRF but removes the directory only if the directory is empty
7. **CD** – The client stores the root of the directory structure in its cache. As the CD command is issued, a search is performed for the path in the directory structure similar to LS. If the path exists, root is changed to that path. After the CD command is issued, all relative paths are prepended with the path of the root stored in the cache before the command is sent to the server. If the path is absolute, no modification is done.
8. **PWD** – Print the current value of the root stored in the client`s cache

# Implementation Details

Following are the packages and their files as per the structure in source code:-

## Client

The client package contains the following files:-

1. **CommandGenerator.java** – Class that contains the functions which generate a configurable number of client requests to be sent to the server using Zipf distribution on the directory structure.
2. **Client.java** – Class that must be invoked to start a client. If the master is alive and running, this class will establish a socket connection to Master on the configured address and port and will send commands to the server and log the results sent back. It will also use the command generator to generate a Zipf distribution of commands to create uneven load on the master
3. **CephClient.java** – Similar to the client, except it is used to connect to the Ceph solution hosted by the master. This client maintains the closest MDS in its cache and also initiates locking to get the resources
4. **CSVFileWriter**.**java** – Class which implements writing the performance results to a CSV file in append mode. Performance is logged as a row with details of command type, metadata solution type, time taken, depth of the directory in the path, and result of the command.

## Commons

The commons package contains the following files:-

1. **Dir** – This package contains all the classes related to representation of directory and its operations
   1. **Directory.java** – This class provides the representation of the directory structure as a B-tree with [metadata](#_Directory_Metadata) supported as discussed
   2. **DirectoryParser.java** – This class provides the functionality to parse the output of listfs.py (Huang, 2015) and create the directory structure represented by Directory.java
   3. **ICommandOperations.java** – This file declares the interface that defines the signatures of all [commands](#_Commands_Supported) that need to be implemented. Every solution must implement this interface and override the methods to provide their unique implementation and discussed in the [detailed design](#_Detailed_Design).
2. **AppConfig.java** – Class that reads all the configuration files in the “conf” folder in the project directory and creates a map for all the properties
3. **AppWatch.java** – Class that provides the utility to log performance over a section of code
4. **CommandsSupported.java** – File that defines an enumeration of all the commands supported
5. **CompletionStatusCode.java** – File that defines an enumeration to specify all the possible results of command execution on the server
6. **Globals.java** – Class to store all the common shared data
7. **Message.java** – Class that defines a serializable type that can be used to wrap messages used for client-server communication
8. **OutputFormatter.java** – Class to provide functionality to pretty print the output of command execution by the server

## Master

The master package contains the following files and packages:-

1. **Master.java** – This class invokes the master server. In order to start the master server, we need to run this class
2. **Listener.java** – This class is invoked by the master class that creates the connection for the sockets used by each directory solution and then creates threads where each thread is an instance of the worker class
3. **Worker.java** – Class which receives the command from the client and executes it for the specified directory solution. It then returns back the output to the client and waits for the next command
4. **ceph** – This package contains the implementation related to Ceph directory solution
   1. **CephDirectoryOperations.java** – This class implements the ICommandOperations interface and provides the implementation of all commands as discussed in the [Ceph](#_Ceph_Directory_Operations) design
5. **metadata** – This package provides the implementation of the MDS and other metadata information needed by directory solutions
   1. **Inode.java** – Contains the inode information of a node in the directory structure. This includes a unique inode number and list of MDS servers
   2. **MetadataManager.java** – This class provides the functionality to generate and store metadata for GFS and NFS. For Ceph it assigns the inode information and calculates the cluster map for MDS
   3. **MetadataServerInfo.java** – A class to represent and MDS server for Ceph. It stores information about the name and IP address of the server and also the status of the server and whether this server is a primary or replica.
6. **gfs** – This package contains the implementation related to GFS directory solution
   1. **GFSDirectoryOperations.java** – This class implements the ICommandOperations interface and provides the implementation of all commands as discussed in the [GFS](#_GFS_Directory_Operations) design
   2. **GFSMetadataReplicationOperations.java** – This class extends the GFSDirectoryOperations and provides implementation of executing the commands on the GFS replica
7. **nfs** – This package contains the implementation related to NFS directory solution
   1. **NFSDirectoryOperations.java** – This class implements the ICommandOperations interface and provides the implementation of all commands as discussed in the [NFS](#_NFS_Directory_Operations) design
   2. **NFSDirectoryParser.java** – This class extends the DirectoryParser class to provide its own implementation of parsing the directory structure and creating the required cut files. It then places these cut files to the NFS shared folder
8. **tst** – This package hosts all the unit tests for the code
9. **conf** – This directory is the location to place all the configuration files related to the project
10. **data** – This directory is the location to store all the 3rd-party libraries which are required in the build path of the project
11. **logs** – This is an empty directory where all the logs of the system are generated.

## Analysis

This folder contains the R scripts to generate plots and graphs from the experimental results of performance. It contains the following files:-

1. **performance.csv** – For each file solution a performance CSV file exists that contains the results of the study on that solution
2. **GeneratePerformance.R** – R script to generate plots and graphs from the CSV files

# Problems Encountered

The problems encountered were as follows:-

1. **Understanding DHT solution** – There was a confusion initially around DHT implementation with level cuts which was sorted out after discussions with professor and TA
2. **Configuration of static IP address in virtual machines** – With the systems setup in the lab, we faced initial difficulties configuring a bridged network between physical hosts and the VMs with static IP addresses. This was resolved after a lot of troubleshooting and help from documentation on the internet
3. **Configuration of log4j in clusters** - With the systems setup in the lab, we faced initial difficulties configuring the master and client to generate log files in which the output of the applications were written. This was resolved after a lot of troubleshooting and help from documentation on the internet
4. **Change from DHT to NFS** – As the DHT file system provided to us had issues, the implementation for DHT directory solution did not work. Therefore, we had to switch to the NFS solution late in the project cycle
5. **Delay in obtaining the code/solution** – For the code provided by the TA for the request generator and socket programming solution, we obtained the code for same at a very late stage. By then, we had implemented our own solution for the same and understanding the provided code and modifying it to fit in the project was a cumbersome task and was avoided.
6. **Scaling up to large filesystems** – The project was designed to parse only up to 500MB of directory structure file, especially for GFS as it wrote all metadata in a single file. This issue was resolved for NFS by increasing the JVM as NFS stores data in files only and needs memory only for parsing. For Ceph, we can have more MDS so that the partition size is always within reasonable limits. But for GFS, since all the data is kept in memory, scaling up became an issue after 500MB
7. **End-to-end integration** – As we worked on different modules in isolation, there were time-consuming issues and bugs encountered when integrating the code and testing end-to-end functionality

# Experimental Results

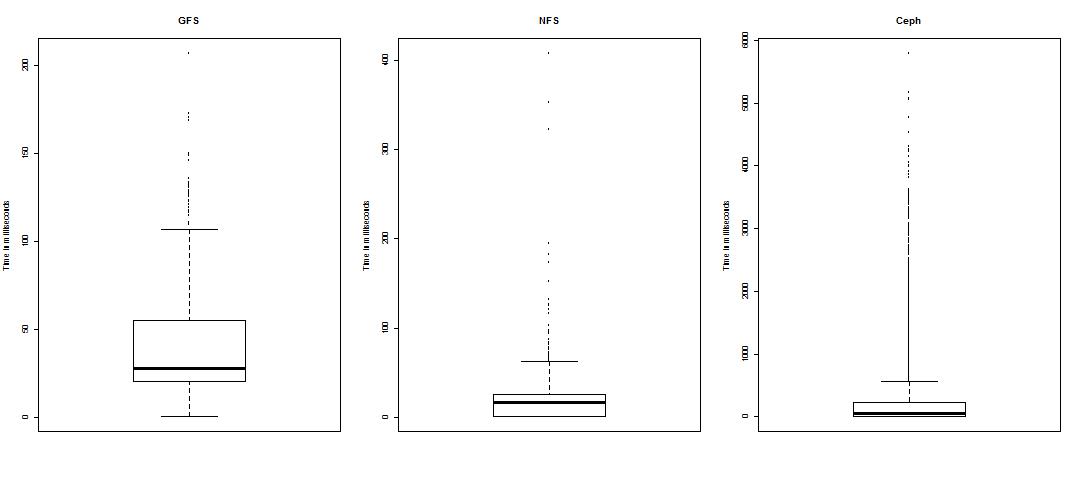
Execution of each command on the server was recorded for performance on a scale on nanoseconds. Following parameters where chosen to depict the variation of runtimes:-

1. Performance compared with depth of the node in the directory structure
2. Performance compared to other commands
3. Performance compared to valid and invalid commands
4. Performance compared to same command on other directory solutions

Results of these above comparisons are as follows:-

### Performance by each solution

Following is the plot to compare the overall performance of each solution:-

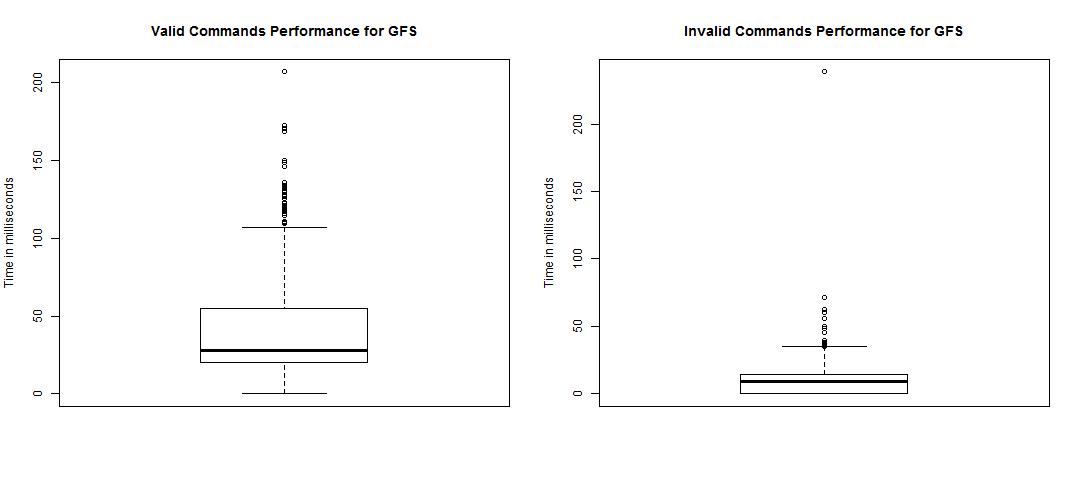


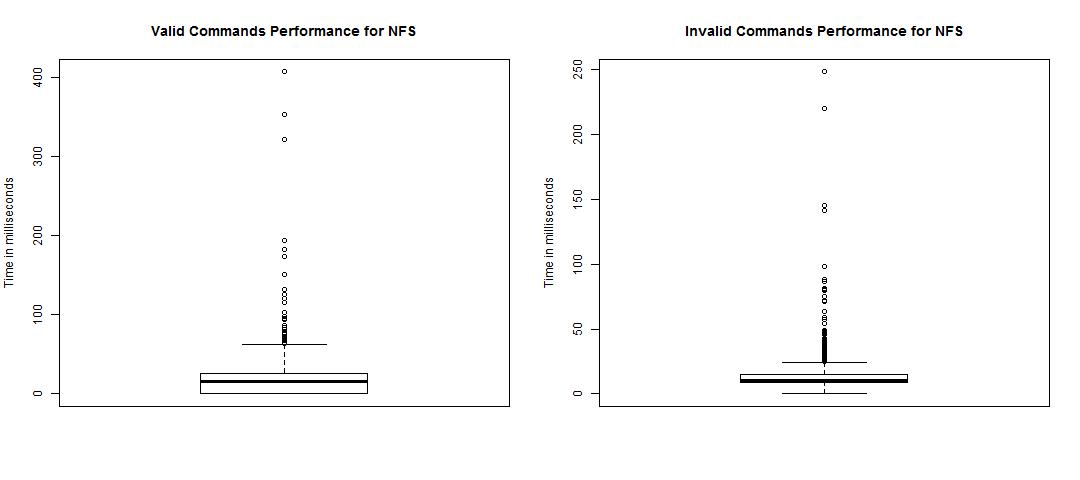
From the above plot, it can be observed that:-

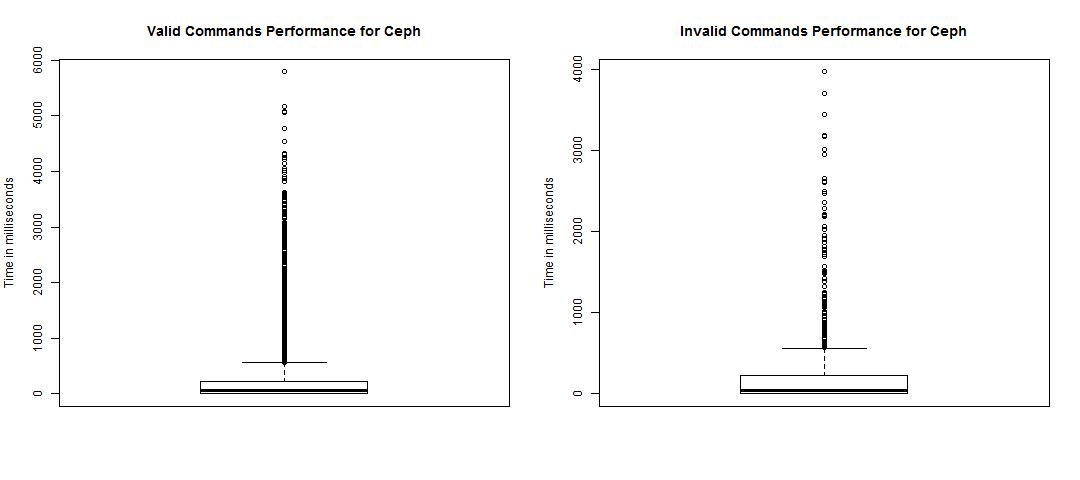
1. Ceph performs the worst as it is dependent on network to acquire locks and caching
2. GFS performs slightly better than NFS as search in GFS happens in a B-Tree for which the worst case complexity of search is better than linear search which happens in a cut file for NFS. However if the cut file is too small, NFS wins over GFS
3. NFS has a lot more outliers depicting that a very large cut file was queried, on which linear search took much more time than the same command in GFS

### Performance by valid and invalid

Following are the plots for each file solution to compare their performance by valid and invalid commands:-





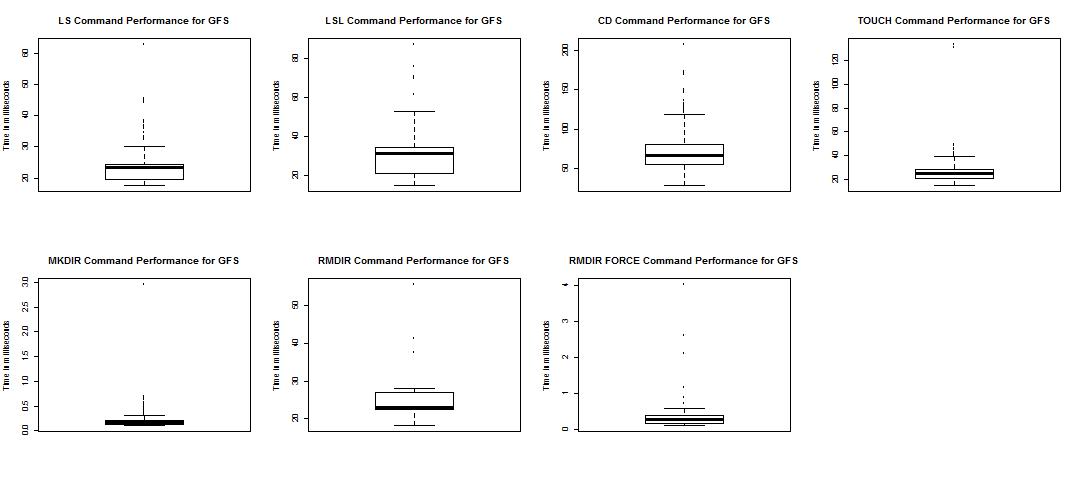


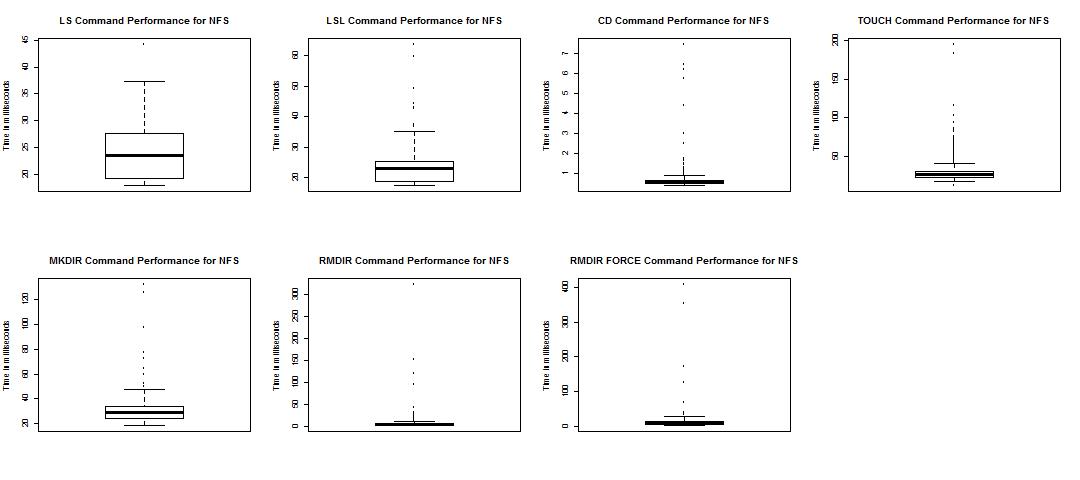
From the above plots, it can be observed that:-

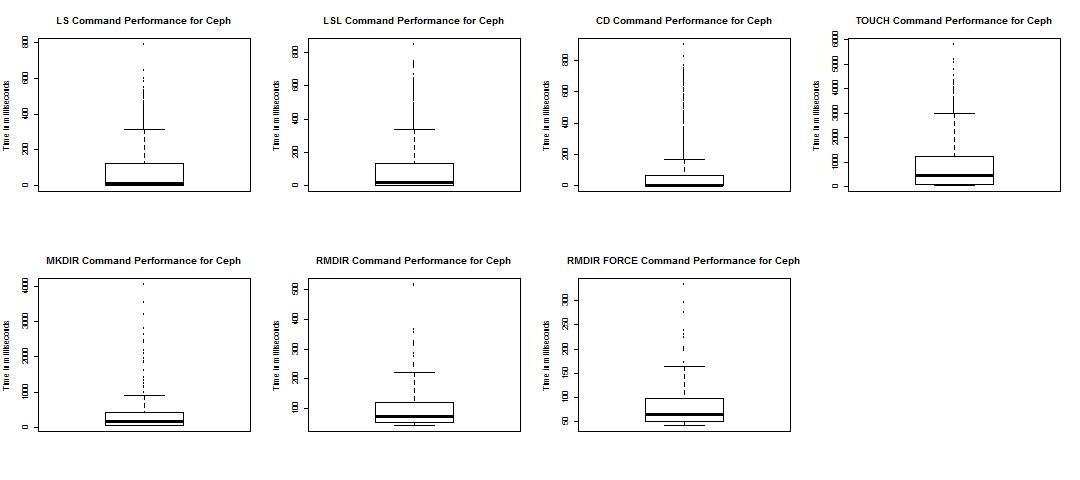
1. Valid commands take more time to execute than invalid commands because valid commands take more time to compose results and return them to the client. Also invalid commands may end prematurely depending upon the checks
2. As stated previously, Ceph performs the worst due to dependencies on network. GFS has a slightly better performance than NFS due to the faster search in B-Tree compared to linear in NFS cut files

### Performance by each command type

Following are the plots for each solution comparing performance for each command type:-





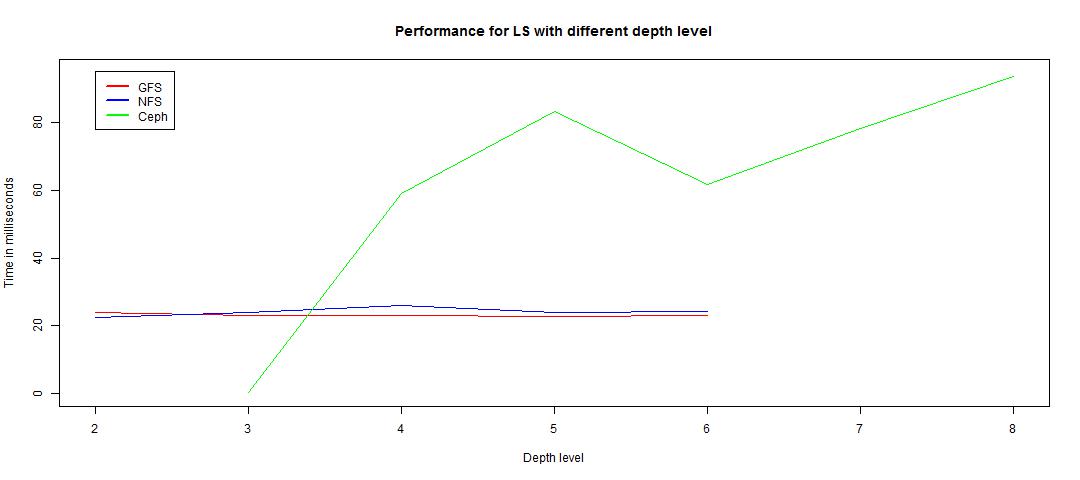


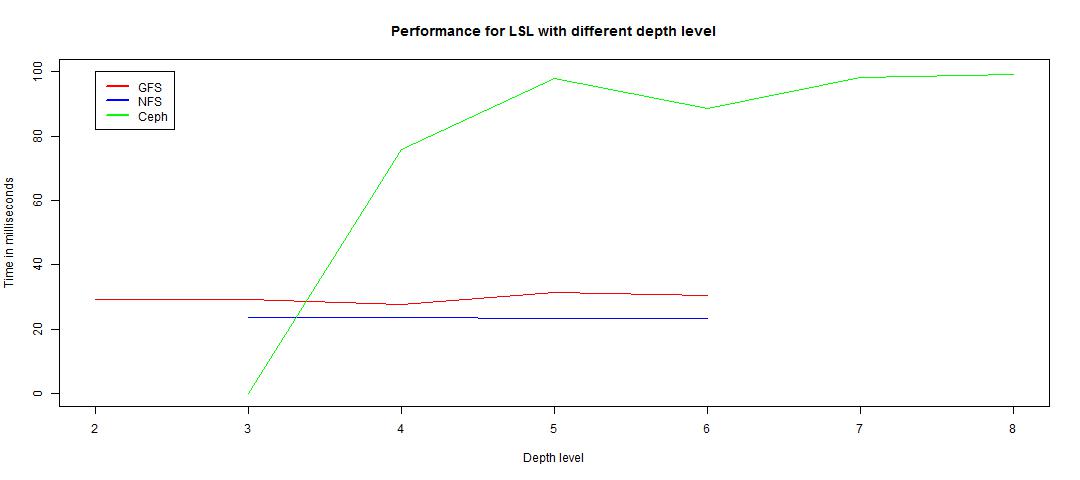
From the above plots, it can be observed that:-

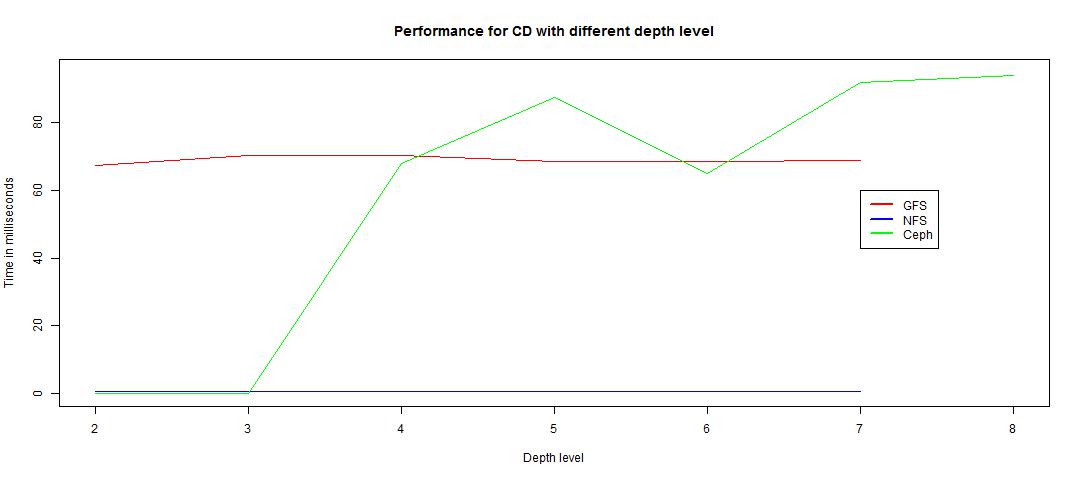
1. LS performs slightly better than LSL because LSL has to compose and return more data to the client
2. CD usually performs better than LS, but due to randomness in the commands for the test data, commands chosen for CD were much deeper than the commands for LS
3. TOUCH performs worse than MKDIR for GFS and NFS because it has to update the timestamp of the file if it already exists, but for MKDIR only a pointer is created to a new node. For Ceph the performance of TOUCH is better than MKDIR because the new directory created might be placed in a different MDS.
4. For Ceph and GFS, RMDIRF performs better than RMDIR because RMDIRF in GFS does a simple dereferencing of the pointer and for Ceph deletes the references recursively on each MDS. But for NFS, RMDIR performs better because RMDIRF has to recursively check the cut files and linearly delete any references
5. CD, TOUCH and RMDIRF are the best performing commands due to the less amount of data they compose and handle

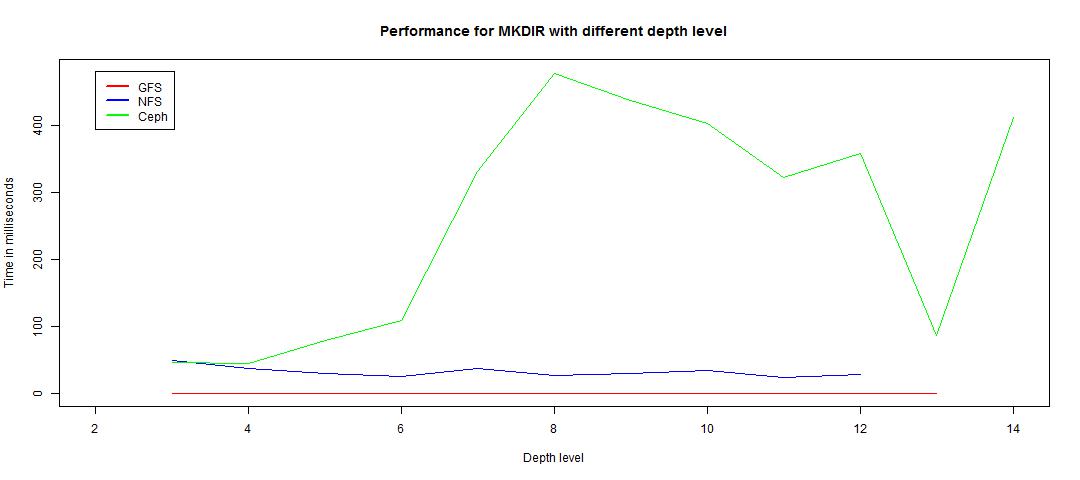
### Performance by node depth

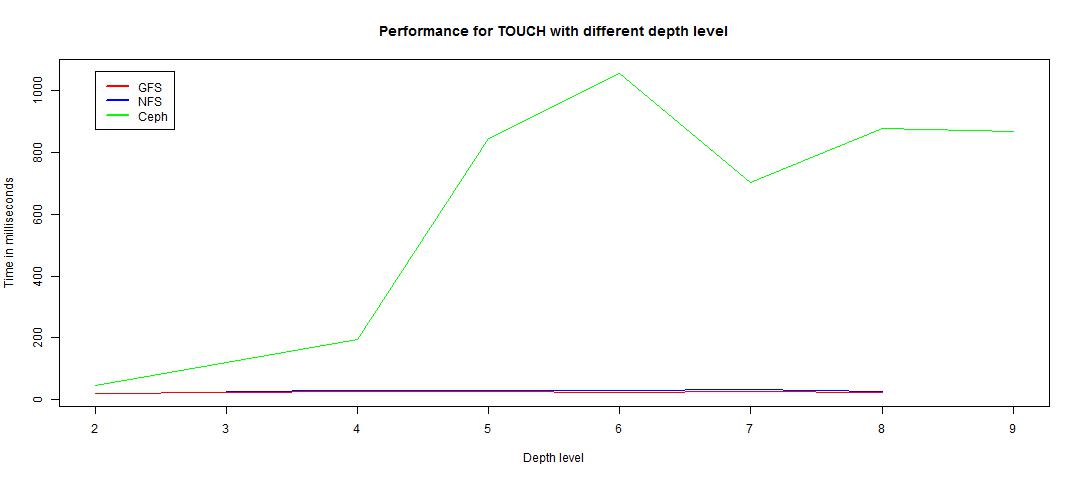
Following are the plots for each commands comparing the performance of each solution with the node depth of the command:-

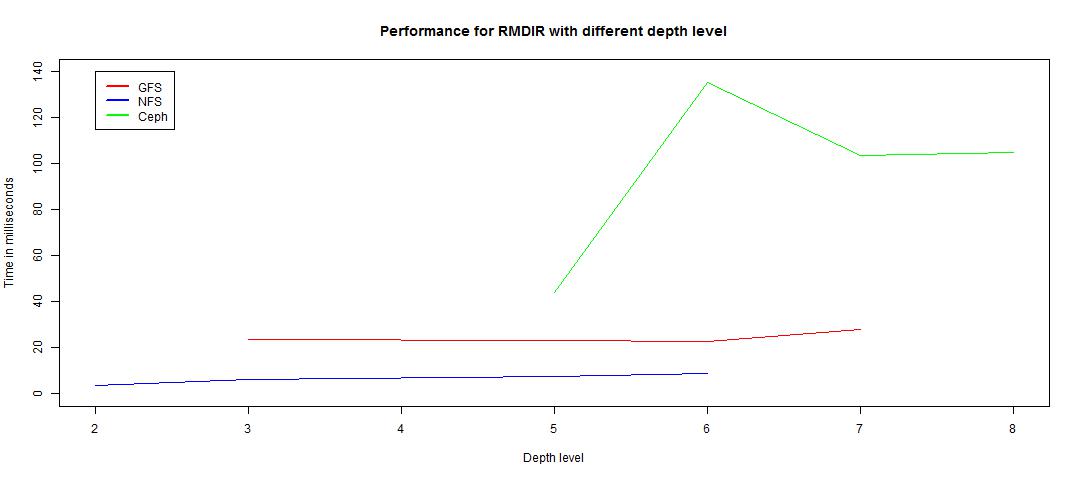


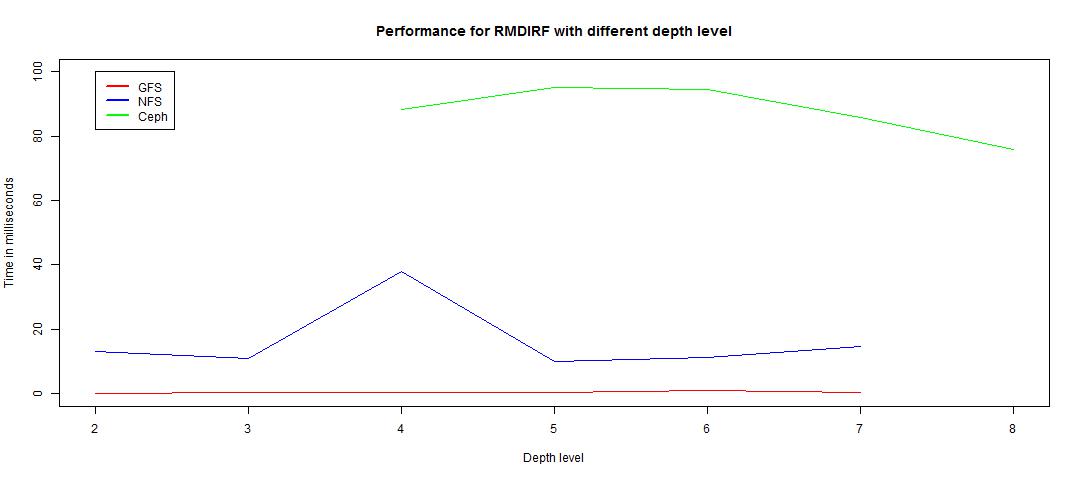












From the above plots, it can be observed that:-

1. Ceph performs the worst except some commands where network communication comes less into play. When network is less involved, ceph has the best performance statistics
2. Performance for LS and LSL is almost same for GFS and NFS. The variation is dependent on the comparison of search in B-tree Vs. linear search in a cut file
3. CD in NFS performs better because most times it checks for existence of a cut file or checks a reference in a small cut file
4. GFS performs better than NFS for MKDIR, TOUCH, RMDIR and RMDIRF because GFS just creates/deletes a reference to a pointer but NFS has to make changes in the cut file

# Installation Manual

Following steps were performed for the installation of setup to run the project:-

1. Cluster setup was – 1 dedicated master machine, 2 shared slave machines for each team
2. Each cluster machine was formatted and CentOS was installed from scratch
   1. CentOS setup file was downloaded from the internet (CentOS, 2015)
   2. The ISO file was loaded into a pen drive and made bootable
   3. This pen drive was used to install the OS in all the machines
3. All the machines were configured to have a static IP
4. Ports for SSH were opened by making an exception in the iptables configuration
5. NoMachine installation file was obtained from the internet (NoMachine, 2015) and was installed on every machine to allow remote login
6. KVM was also installed on each machine to enable hosting of VMs using the documentation from the internet (Wiki, 2015)
7. Similar to setup of each cluster machine, 3 VMs were setup on each physical machine
8. VMs were also configured to have a static IP
9. NFS was setup on the cluster using the documentation from the internet (How To Set Up an NFS Mount on CentOS 6, 2012)

After the setup on the cluster, the local environment was setup to develop the code. The steps are as follows:-

1. A private repository was created on GitHub to host the project`s code
2. Eclipse was chosen as the IDE for code development
3. The libraries in the project data folder were used as helpers to perform trivial tasks in the code
4. There were three run configurations, one for master and the other two for the clients
5. The finished code was exported as three runnable jars, one with client runnable, other for ceph client runnable and third as master runnable
6. Log4j configuration was added to the classpath of the generated jars, so that both client and master are able to generate output in log files
7. These jars were then transferred to the desired cluster machines along with conf, data and logs folder

# User Manual

To use the system, follows the steps as mentioned below:-

1. First the master server should be up and running
   1. Ensure that the configurations are as desired.
   2. The configuration for master server is in conf folder under the file masternode.conf.
   3. For the Ceph configuration, the MDS server configuration is in conf folder under file mds.conf
   4. For the logging configuration, check the configuration in the conf folder under the file log4j.properties
   5. Run the master with the command “java –jar Master.jar”
2. Ensure the master loads up without any errors
3. Start the client with the desired configuration
   1. Set the client configuration to the IP address of the master and the port of the directory to which the client needs to connect
   2. For the client configuration, check the configuration in the conf folder under the file client.conf
   3. For the logging configuration, check the configuration in the conf folder under the file log4j.properties
   4. For NFS and GFS, run the client with the command “java –jar Client.jar”
   5. For Ceph, run the client with the command “java –jar CephClient.jar”
4. Client will generate random commands using Zipf distribution and send them to the server. The server will respond back with the output
   1. To observe the output of the master, check master logs at the logs folder under the file master.log.out
   2. To observe the output of the client, check master logs at the logs folder under the file client.log.out
5. [Optional] If there is a need to send specific commands by user interaction, change the desired client`s code to taking System.in instead of a file in the executeCommands method.
   1. Generate the client jars again
   2. Run the client
   3. Type in a command and observe the client logs for output

# Workload Distribution

### Summary

#### Ekal

1. Ceph and GFS design
2. GFS implementation
3. DHT implementation and debugging the DHT code provided by TA
4. Utility classes for configuration, performance logging and output formatting
5. Command generation by client using Zipf distribution
6. Midterm and final report
7. NFS locking and bug fixes
8. Generating plots and graphs in R with experimental results

#### Jayakarthigayan

1. Ceph design
2. Initial cluster setup
3. Ceph implementation with locking and replication
4. Ceph client
5. Utilities like completion status codes, MDS manager
6. Configuration of static IP for VM in cluster setup
7. Testing and debugging deployed code on clusters
8. Experimental study and performance results

#### Ketan

1. GFS design
2. Initial cluster setup
3. GFS implementation with locking and replication
4. Client implementation
5. Master implementation with multithreading
6. Utilities like global shared data, serializable message type for communication
7. NFS setup on cluster
8. Testing and debugging deployed code on clusters
9. Fixes to scale up to parse the large files provided by the TA

#### Sahith

1. DHT design
2. Initial cluster setup
3. DHT implementation
4. NFS setup on cluster
5. NFS implementation
6. Performance results

### Detailed list

#### Ekal

|  |
| --- |
| **Week 1 Tasks: Sep 13, 2015 to Sep 19, 2015** |
| Read papers and Worked on Design |
| Downloading and making bootable pen drives for installation of OS and VMs |
| Created github repository, tracked machine details and documentation |
|  |
| **Week 2 Tasks: Sep 20 to Sep 26** |
| Helped in discussions for Ceph design |
| Created basic class and packages for initial implementation |
| Added DHT-fs to the project |
| Implemented commands: ls, mkdir, touch |
|  |
| **Week 3 Tasks: Sep 27 to Oct 3** |
| Helped in discussions for Ceph design |
| Testing client and master communication |
| Debugging issues |
| Modified client and server to read configuration from a file |
| Added unit tests |
|  |
| **Week 4 Tasks: Oct 4 to Oct 10** |
| Implemeted basic structure to provide all the command operations |
| Read about Zipf |
| Added implementation to print command output in table format |
| Integrated the socket programming utilities provided by TA to the project code |
|  |
| **Week 5 Tasks: Oct 11 to Oct 17** |
| Read about Zipf |
| Implemented zipf distribution for the clients |
| Implemented command generation for clients in a configurable way |
| Tested and added unit tests for the same |
|  |
| **Week 6 Tasks: Oct 18 to Oct 24** |
| Updated code to use correct version of dht-fs |
| Worked on project report |
| Added support for logging using log4j for client and master |
| Testing end-to-end functionality for midterm and bugfixes |
|  |
| **Week 7 Tasks: Oct 25 to Oct 31** |
| Setting up the hosts and VMs in lab to test midterm expectations |
| Testing and fixes to DHT and GFS code after testing setup in lab |
| Testing scripts provided by TA to list directory structure |
|  |
| **Week 8 Tasks: Nov 1 to Nov 7** |
| Updating project report according to new guidelines |
| Research on which 3rd party libraries to use to log performance |
|  |
| **Week 9 Tasks: Nov 8 to Nov 14** |
| Implementation of performance monitoring in the project |
| Unit tests for performance monitoring code |
| Design and testing of locking model in GFS to allow multiple client to work with server |
|  |
| **Week 10 Tasks: Nov 15 to Nov 21** |
| Remove DHT-FS code and change DHT references to NFS |
| Remove IOControl lib code provided by TA and use our own implementation |
| Completed final report except the experimental results part |
|  |
| **Week 11 Tasks: Nov 22 to Nov 28** |
| Implement locking for NFS and bugfixes |
| Write R code to generate plots and graphs on performance results |
| Update client and master to log performance results |
|  |
| **Week 12 Tasks: Nov 29 to Dec 6** |
| Updated report with experimental results |
| Testing of integrated code in the cluster and locally |

#### Jayakarthigayan

|  |
| --- |
| **Week 1 Tasks: Sep 13, 2015 to Sep 19, 2015** |
| Read papers and Worked on Design |
| Setup Development Environment (OS and VMs) |
|  |
| **Week 2 Tasks: Sep 20 to Sep 26** |
| Installed CentOS 6 on all the nodes |
| Configured ssh |
| Configured VM |
| Assigned static IP to hosts |
| Worked on refining the CEPH design to include the following: 1.failure detection & recovery,  2.fault tolerance,  3.client communications,  4.dynamic partitioning,  5.Authority & Collaborative caching mechanism 6.Initial communication & detail sharing 7.Inode & Directory information finalization |
| Started modifying the current code to support subtree partitioning. |
|  |
| **Week 3 Tasks: Sep 27 to Oct 3** |
| Implemented the basic interface for ceph |
| Implemented MDS server with basic directory structure |
| Modified Data model to include all the information required for CEPH funtionality |
| Implemented code to handle unique inodenumber ranges for each MDS |
| Completed the entire data model required to handle CEPH system. |
|  |
| **Week 4 Tasks: Oct 4 to Oct 10** |
| Implemented LS for CEPH |
| Implemented partitioned file search api to search across multiple MDS |
| Implemented socket communication functionality to perform remote command execution |
| Started implementing MKDIR for directories residing locally. |
|  |
| **Week 5 Tasks: Oct 11 to Oct 17** |
| Completed MKDIR implementation for directories spanning across multiple MDS |
| Fixed various issues occurring in LS and MKDIR commands while executing them in remote MDS |
| Completed TOUCH implementation for files and directories residing in local and also in remote MDS |
| Fixed firewall issues that are occurring in the lab machine while communicating across MDS (i.e across hosts and VMs) |
|  |
| **Week 6 Tasks: Oct 18 to Oct 24** |
| Completed RMDIR command implementation for CEPH |
| Fixed issues occurring in TOUCH command for existing files. |
| Completed replicas MDS update process for all the commands involving metadata update. |
| Completed implementing a fully configurable MDS Directory partition operation. |
| Completed implementing a initiator to communicate the initial Directory partition to all respective MDS and replicas |
| Implemented the code to identify the directories/nodes that are heavily used in each MDS based on the workload counter associated with each node/directory. This measures the overall workload for each MDS too. |
|  |
| **Week 7 Tasks: Oct 25 to Oct 31** |
| Set up the hosts and VMs in lab to test midterm expectations |
| Tested and fixed CEPH code after testing setup in lab |
|  |
| **Week 8 Tasks: Nov 1 to Nov 7** |
| Completed implementing CD and PWD commands for CEPH File System. |
| Working on designing a locking system for CEPH to support locking across multiple threads. |
|  |
| **Week 9 Tasks: Nov 8 to Nov 14** |
| Completed implementeing LSL i.e ls -l for the CEPH File System |
|  |
| **Week 10 Tasks: Nov 15 to Nov 21** |
| Created a separate client module for CEPH File system since ceph client is responsible for more functionalities. |
| Implemented and Tested CEPH client caching. |
|  |
| **Week 11 Tasks: Nov 22 to Nov 28** |
| Completed implementing RMDIRF i.e rmdir -f command for CEPH file system to scale across multiple MDS and also replicas update. |
| Completed implementing a locking system for CEPH file system which can span across different threads. |
| Coded the CEPH locking system to work across multiple primary MDS |
| Changed the ceph client code to trigger locking and unlocking based on the commands executed. |
| Tested various commands over CEPH file systems and fixed the bugs in various scenarios scaling single MDS, Multiple MDS and replica update support |
|  |
| **Week 12 Tasks: Nov 29 to Dec 6** |
| Fixed performance bugs and geenrated reports for various cases of CEPH |
| Testing of heavy load of data over multiple nodes and replicas |

#### Ketan

|  |
| --- |
| **Week 1 Tasks: Sep 13, 2015 to Sep 19, 2015** |
| Read papers and Worked on Design |
| Downloading and making bootable pen drives for installation of OS and VMs |
| Created github repository, tracked machine details and documentation |
|  |
| **Week 2 Tasks: Sep 20 to Sep 26** |
| Participated in discussions for centralized approach and Ceph design |
| Started implementation - creation of basic classes, packages |
| Did coding for network communication and thread management for master |
| Created parser for reading existing directory structure |
| Implemented network communication and thread management for master |
|  |
| **Week 3 Tasks: Sep 27 to Oct 3** |
| Added directory metadata info - access rights, timestamp, size |
| Added command execution capability to client |
| Implemented rmdir on master |
| Corrected input file parser |
| Serialize and store the metadata to avoid parsing every time master is up |
| Correct the master client commnication - ls working |
| Designed and discussed ideas for client requests |
|  |
| **Week 4 Tasks: Oct 4 to Oct 10** |
| Implemeted basic structure to provide all the command operations |
| Read about Zipf |
|  |
| **Week 5 Tasks: Oct 11 to Oct 17** |
| Read about Zipf |
| Fix GFS working and test the commands |
| Design replication for GFS metadata |
|  |
| **Week 6 Tasks: Oct 18 to Oct 24** |
| Worked on project report |
| Make metadata cloneable |
| Implemented replication for GFS metadata |
| Corrected mkdir and touch implementation for GFS |
| Splitting and serializing the metadata based on MDS partitions |
| Testing and debugging DHT approach |
|  |
| **Week 7 Tasks: Oct 25 to Oct 31** |
| Testing and fixes to DHT and GFS code after testing setup in lab |
| Testing scripts provided by TA to list directory structure |
| Modifying the existing parser for new input format |
|  |
| **Week 8 Tasks: Nov 1 to Nov 7** |
| Research on which 3rd party libraries to use to log performance |
|  |
|  |
| **Week 9 Tasks: Nov 8 to Nov 14** |
| Implementation of performance monitoring in the project |
| Discuss the design for locking model in GFS |
| Implement and test locking in GFS |
| Implement PWD, CD, LSL (ls -l), RMDIRF (rmdir -f) commands |
|  |
| **Week 10 Tasks: Nov 15 to Nov 21** |
| Setup and configure cluster and test working |
|  |
| **Week 11 Tasks: Nov 22 to Nov 28** |
| Implement NFS solution for DHT approach |
| Create NFS shares on physical machines |
| Mount the share on the VMs and configuration |
|  |
| **Week 12 Tasks: Nov 29 to Dec 6** |
| Test the new data providedd by TA |
| Changes made in the code to work with the new metadata format |
| Test performances for Ceph NFS and GFS |

#### Sahith

|  |
| --- |
| **Week 1 Tasks: Sep 13, 2015 to Sep 19, 2015** |
| read papers on GfS and Ceph and worked on design |
| Installed OS and VM's on the machines |
|  |
| **Week 2 Tasks: Sep 20 to Sep 26** |
| Installed CentOS 6 on all the nodes,Configured SSH |
| helped assigning static IP to hosts |
| Helped in discussions for Ceph and GFS design |
|  |
| **Week 3 Tasks: Sep 27 to Oct 3** |
| Read papers on DHT and started designing |
| Discussed on how to generate mix of client |
| requests to access directories |
| Client side parsing of directory structure |
|  |
| **Week 4 Tasks: Oct 4 to Oct 10** |
| Created parser for reading existing directory |
| structure and created files based on cut. |
| Implented ls,rmdir commands for DHT file system. |
|  |
| **Week 5 Tasks: Oct 11 to Oct 17** |
| Implented rmdirf,mkdir,touch commands for DHT |
| file system. |
| Debugging Issues |
|  |
| **Week 6 Tasks: Oct 18 to Oct 24** |
| Helped integrating DHT\_FS with our dht Operations |
| and tried placing files in DHT\_FS. |
| Helped in deploymment and testing of code on nodes. |
| Bug fixes |
|  |
| **Week 7 Tasks: Oct 25 to Oct 31** |
| Testing scripts provided by TA to list directory structure |
| Testing and fixes to DHT and GFS code after testing setup in lab |
|  |
| **Week 8 Tasks: Nov 1 to Nov 7** |
| Map reduce Assignment |
|  |
| **Week 9 Tasks: Nov 8 to Nov 15** |
| Waiting for TA's code for DHT File system |
| Implemented LS-L,CD and PWD for DHT file system. |
|  |
| **Week 10 Tasks: Nov 16 to Nov 23** |
|  |
| **Week 11 Tasks: Nov 23 to Nov 30** |
| Implement NFS solution for DHT approach |
| Created and shared a common directory to all VMs |
| Mount the NFS directory on all VMs |
| Implemented and tested locking on DHT |
|  |
| **Week 12 Tasks: Nov 1 to Nov 7** |
| Change the code and test the data provided by TA in Cluster |

# References

Apache. (2012, May 13). *Log4j*. Retrieved from Logging: https://logging.apache.org/log4j/1.2/download.html

Apache. (2014, Feb 17). *IO*. Retrieved from Commons: https://commons.apache.org/proper/commons-io/download\_io.cgi

Apache. (2014, July 18). *Logging*. Retrieved from Commons: http://commons.apache.org/proper/commons-lang/download\_lang.cgi

Apache. (2015, May 18). *Lang*. Retrieved from Commons: http://commons.apache.org/proper/commons-lang/download\_lang.cgi

Apache. (2015, Sep 11). *Math*. Retrieved from Commons: http://commons.apache.org/proper/commons-math/download\_math.cgi

Apache. (2015, Aug 2). *Pool*. Retrieved from Commons: https://commons.apache.org/proper/commons-pool/download\_pool.cgi

*CentOS*. (2015). Retrieved from Download: https://www.centos.org/download/

Google. (2003). *Google File System.* Retrieved from http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

Google. (2009, June 3). *Collections*. Retrieved from Maven: http://central.maven.org/maven2/com/google/collections/google-collections/1.0-rc2/google-collections-1.0-rc2.jar

Google. (2010, June 1). *Guava*. Retrieved from Maven: http://central.maven.org/maven2/com/google/guava/guava/r05/guava-r05.jar

*How To Set Up an NFS Mount on CentOS 6.* (2012, September 18). Retrieved from DigitalOcean: https://www.digitalocean.com/community/tutorials/how-to-set-up-an-nfs-mount-on-centos-6

Huang, Y. (2015, November). *Announcements*. Retrieved from E-learning: Listfs.py

*Ini4j*. (n.d.). Retrieved from Sourceforge: http://downloads.sourceforge.net/project/ini4j/ini4j-bin/0.5.5-SNAPSHOT/ini4j-0.5.5-SNAPSHOT-bin.zip

*NoMachine.* (2015). Retrieved from Download: https://www.nomachine.com/download

Sun Microsystems. (1984). *NFS*. Retrieved from Stanford: http://web.stanford.edu/class/cs240/readings/nfs.pdf

University of California. (2006). *Ceph: A Scalable, High-Performance Distributed File System.* Retrieved from http://www.ssrc.ucsc.edu/Papers/weil-osdi06.pdf

*Wiki.* (2015, May 9). Retrieved from CentOS: https://wiki.centos.org/HowTos/KVM

Yen, I. L. (2015). *Bigdata.* Retrieved from Class notes: E-learning

Yen, I. L. (2015, 10 12). *Cloud Computing.* Retrieved from I-Ling`s home page: http://www.utdallas.edu/~ilyen/course/cloud/for15f/term-proj-midrep.pdf