GROUP C1

Ekal Golas, Jayakarthigayan Sridharan, Ketan Joshi, Sahith Katukuri

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

MIDTERM REPORT

CLOUD COMPUTING PROJECT



Table of Contents

[Group Information 2](#_Toc433389954)

[Details of team members 2](#_Toc433389955)

[Problem Statement 3](#_Toc433389956)

[System architecture 4](#_Toc433389957)

[High-Level Design 5](#_Toc433389958)

[Detailed Design 6](#_Toc433389959)

[Ceph 6](#_Toc433389960)

[GFS 6](#_Toc433389961)

[DHT 7](#_Toc433389962)

[Implementation Details 8](#_Toc433389963)

[Client 8](#_Toc433389964)

[Commons 8](#_Toc433389965)

[Master 9](#_Toc433389966)

[GFS Directory Operations 9](#_Toc433389967)

[Ceph Directory Operations 9](#_Toc433389968)

[DHT Directory Operations 11](#_Toc433389969)

[Problems Encountered 12](#_Toc433389970)

[References 13](#_Toc433389971)

# 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

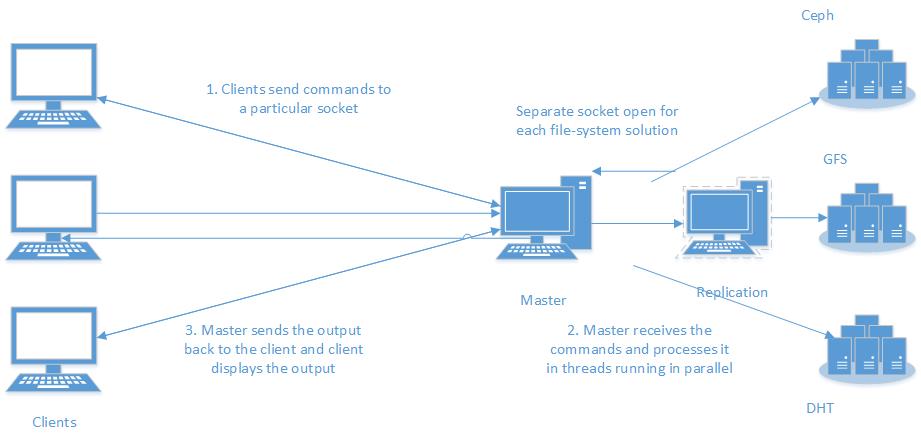
## Functionalities provided

Following will be the functionalities provided with the project:-

1. Independent 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. Logging to track all requests and responses
2. Independent 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 DHT 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. 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
   4. Support for commands on each directory solution – 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
3. GFS solution – This creates a B-tree structure in memory with replication
4. Ceph solution -

# System architecture

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



# High-Level Design

As discussed in the problem statement (Yen, Cloud Computing, 2015), the design for the different solutions and the overall implementation is stated as follows:-

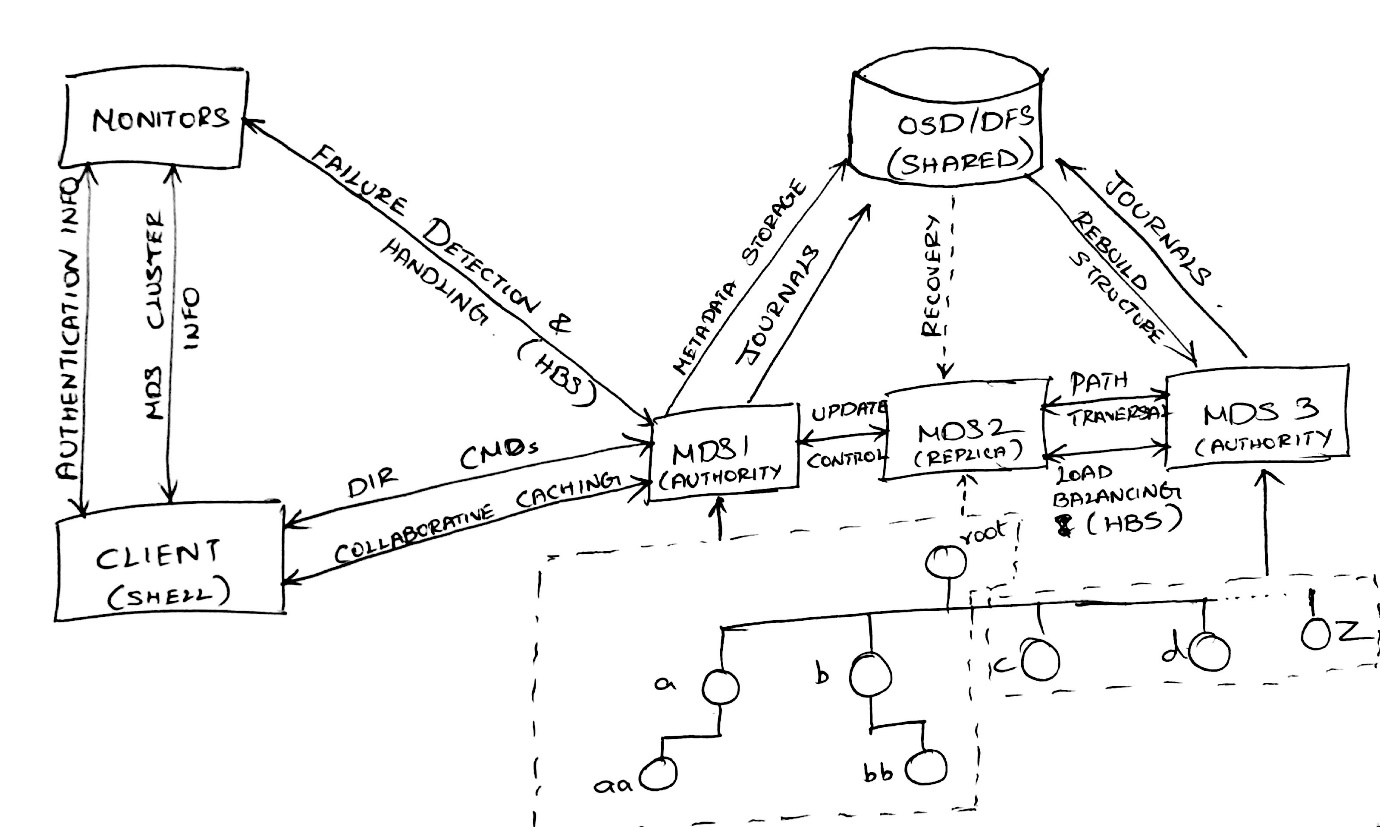
There are three components to the system, each represented as packages which represent the following:-

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. DHT 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. Code provided by the TA for communication (danei, 2015)
   3. Configuration and other utilities

# Detailed Design

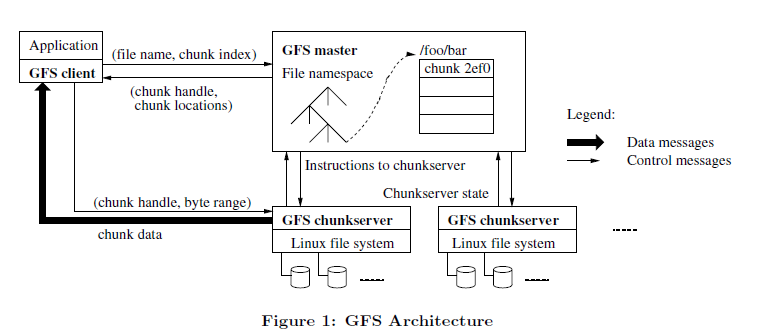
## Ceph

A design was developed using the Ceph paper (University of California, 2006). The design diagram is stated as follows:-



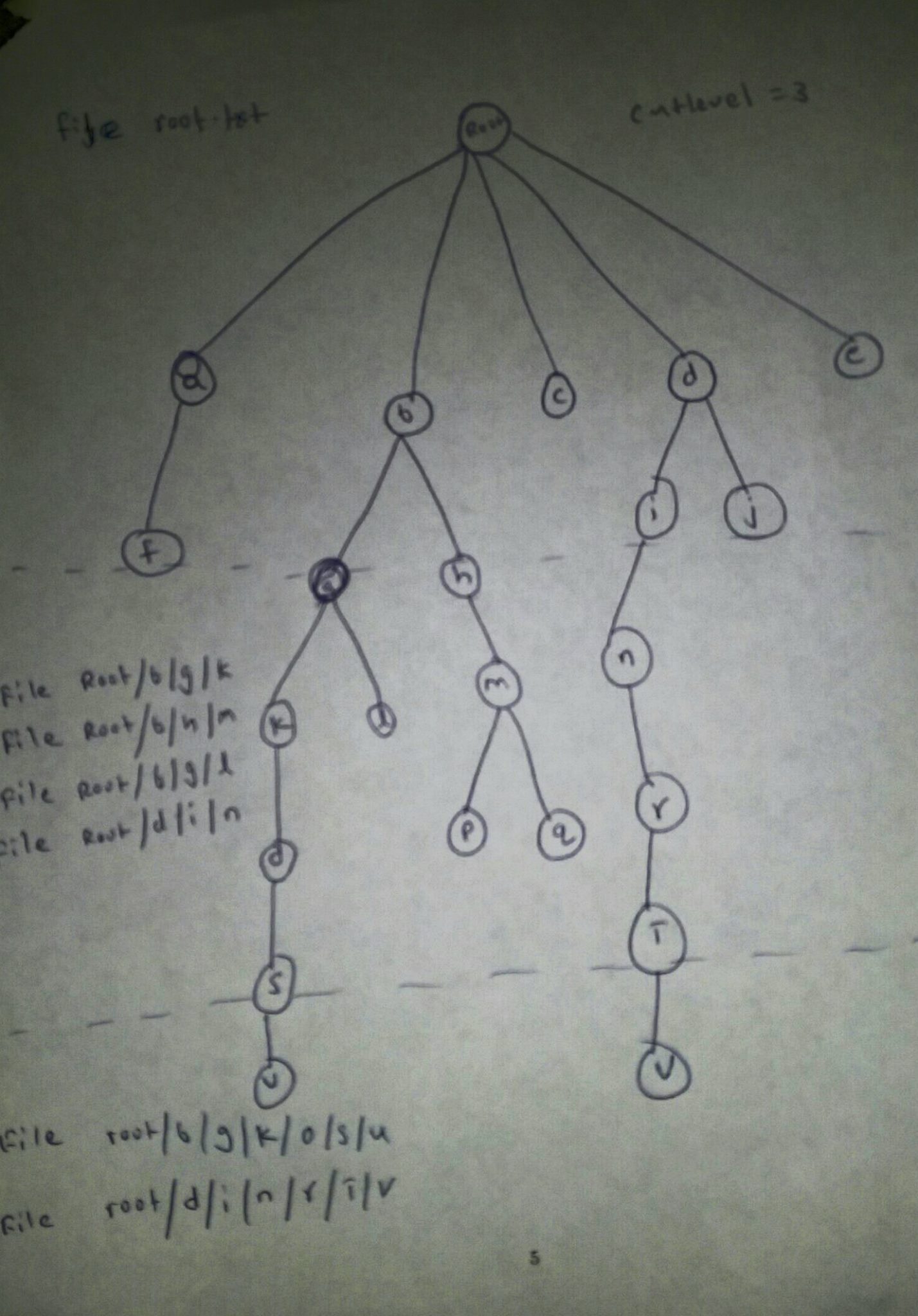
## GFS

A design was developed using the GFS paper (Google, 2003) and the class notes (Yen, Bigdata, 2015). The design diagram is stated as follows:-



## DHT

A design was developed using the DHT filesystem code (Yinzi, 2015) provided by the TA and class notes (Yen, Bigdata, 2015). The design is stated as follows:-



# Implementation Details

## 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. Each node in the tree stores the following information:-
   1. Name of this directory/file
   2. Inode
   3. Access rights
   4. Timestamp last modified
   5. Whether this node is a file or directory
   6. Operation counter to keep track of number of accesses
   7. Size
   8. List of sub-directories at this node
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 as follows:-
   1. Ls – List directory, just the name and type for now
   2. Touch – Create a file
   3. Mkdir – Create a directory
   4. Rmdir – Remove a directory, recursively for now
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. 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 directory information
2. MKDIR – Performs a search similar to LS to get the parent directory. If LS returns an error, error is returned, else a new child node is created in the parent node as a directory.
3. TOUCH – Similar to MKDIR except that the new child gets added as a file.
4. RMDIR - Performs a search similar to LS to get the parent directory. 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

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 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.
   2. 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.
   3. 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.
   4. If the expected file/directory is found then return the list of details for that node and its children if exists.
   5. 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.
2. 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 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.
   2. 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.
   3. 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.
   4. 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.
   5. 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.
   6. 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.
3. 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 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.
   2. 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.
   3. 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.
   4. 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.
   5. 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.
   6. 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.
   7. 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.
4. 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 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.
   2. 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.
   3. 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.
   4. 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.
   5. 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.
   6. 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.
   7. 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.

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.

## DHT Directory Operations

The commands for DHT are implemented as follows:-

1. LS – It takes the file path as an argument and split the file path based on the level cut number and takes the trimmed path and search in the Hash map for the corresponding file and read line by line to look for corresponding path information and output the lines which matches the path.
2. MKDIR – Performs a search similar to LS to get the corresponding file and search for the line which contains the given path if found throw an exception saying that directory exists else write a line with the given path and if it is at the level cut number create a file with the full path and insert in the Hash map.
3. TOUCH – Similar to MKDIR search for the path in the file if found then rewrite the line with the updated time stamp else append the path in the file. `
4. RMDIR - Perform a search similar to LS and get the file and remove the lines which matches the pathname and also remove the files whose subdirectories information is stored in different files by searching in the Hash map.

# Problems Encountered

The problems encountered so far are 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. 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

# 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

danei. (2015). *Request-response interface*. Retrieved from Bitbucket: https://bitbucket.org/danei/iocontrol/

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

*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

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

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

Yinzi. (2015). *DHT file system*. Retrieved from Bitbucket: https://bitbucket.org/utdcloudlab/dht-fs