Multi-User Distributed Cloud File System

Nipesh Roy
Northeastern University
College of Computer and Information Science

Sharmodeep Sarkar
Northeastern University
College of Computer and Information Science

Soham Sankhe
Northeastern University
College of Computer and Information Science

Sanket Navalakha
Northeastern University
College of Computer and Information Science

ABSTRACT

File system in User space (FUSE) is an operating system mechanism for Unix-like computer operating systems that lets non-privileged users create their own file system formats without editing kernel code. Multi-User Distributed Cloud File System is a flexible file system framework based on FUSE for concurrent and synchronized usage by multiple authenticated users wherein the entire file storage is abstracted onto multiple Dropbox cloud spaces. The system is extremely easy to install onto any UNIX system with minimal pre-requisites (basic Java and Python). It ensures multi user data consistency. Multi-User Distributed File System is extremely easy to install, use as well as add multiple cloud storages in the background to store data on the fly.

Keywords— FUSE, MUTDS, cloud, multi-user, consistency

CONTRIBUTION

	Sharmo	**Nipesh**	**Soham**	**Sanket**
FUSE Research	35 %	35%	15%	15%
FUSE customiza-tion	35%	35%	15%	15%
Backend Research	20%	20%	30%	40%
Backend Implemen-tation	10%	10%	40%	30%

INTRODUCTION

Nowadays thousands of companies have shifted their data storage to cloud. Although there are security concerns with storing data in cloud, but everyday better solutions are being proposed which is making the storage on cloud safer and secure to implement a new file system, a handler program linked to the supplied libfuse library needs to be written. The main purpose of this project is to specify how the file system is to respond to read/write/stat requests. The program is also used to mount the new file system. [1] When the file system in user space issues a mount request it gets registered in the kernel. If a user now issues read/write/stat requests over this newly mounted file system, FUSE sends these calls to the custom implementation to handle different back end stores in our case Cloud based storage such as Dropbox, Google Drive and One Drive etc.

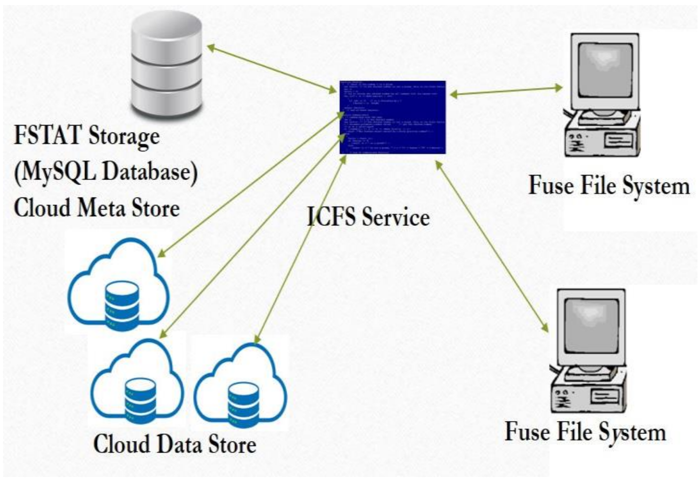


Fig A: Multi-User Distributed Cloud File System

ARCHITECTURE DESIGN

The overall architecture of Multi-User Distributed Cloud File System consists of the following components

1. FUSE for Cloud FS.
2. Remote File metadata server.
3. Cloud based data-storage (Dropbox, One Drive, etc.).

A. FUSE for Cloud FS

The FUSE implementation for Cloud FS is developed in python using fusepy. It offers almost every file system operation those are required to be provided by a File System module in user space to bypass traditional VFS and redirect these calls to cloud and also provides all user applications for the fuse mount. The architecture consists of a remote database so which makes the solution easily portable. To avoid this expensive operation for which involves obtaining metadata for the files stored in the cloud our implementation of Cloud Fuse provides a caching mechanism which is based on a time-to-live mechanism which invalidates the cache entries after every 30seconds and also a signaling mechanism which will signal if a cache entry is invalidated during the TTL period.

B. *Remote File metadata server*

For our FUSE system, we needed a simple server that will handle all the functionalities that concern cloud storage and database operations. The MUDFS service is implemented in Java. The reason for implementing the service in Java other than it being a strong server-side technology is that it has a strong support from cloud storage providers in the form of libraries and APIs. The server stores metadata remotely which in turn increases the portability of the system

DB Schema:

The metadata information for files and directories are maintained in the fstat table. We treat files and directories similarly in a way that both will have an entry in the fstat table. The hierarchy is maintained using the filename and file path entries in the fstat table. Operations on files involve connecting with Dropbox whereas directories only result in making an entry in the database similar to Linux Directory entries. For each entry in the fstat table, a new key is generated called the inode number. Every Dropbox account is treated as a device. The metadata for the devices is maintained in the cloud_metadata table. Addition of new devices only requires one entry in the device table in the database.

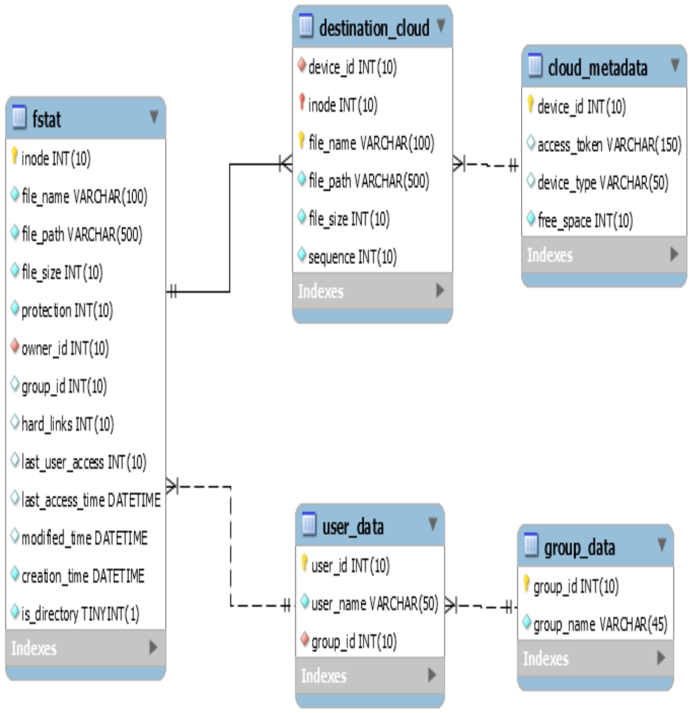


Fig B: EER Diagram for the database

For every file, its inode entry in the fstat table will have a corresponding entry in the destination_cloud table which indicates the device on which the file is stored. If a file is split, it may reside on more than one devices. Thus, the fstat and the destination_cloud table have a cardinality of 1 to many. User and group information is maintained in user_data and group_data tables respectively.

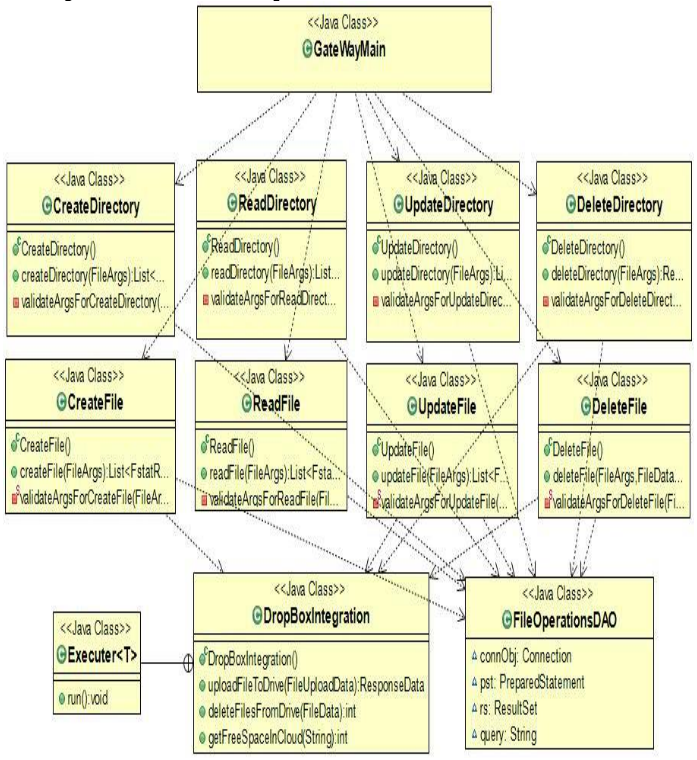
MUDFS (Backend):

We find that all the operations are done by the FUSE system translate into simple CRUD operations with reference to the cloud storage. The service majorly exposes 8 operations to the FUSE system, namely CRUD for files and CRUD for directories.

For create, if the given file does not already exist in the database, we begin with querying the cloud metadata to check if the combined free space is enough for the file to be stored on the cloud. If any single device does not have enough free space, we split the file into smaller chunks and upload them. The device selection depends on which one contains the maximum free space. Once the file is uploaded to Dropbox, we store the relevant information in the database and the resulting fstat information is returned to the client.

File update operations begin with deleting the older version of the file. The new updated file is uploaded to Dropbox. The file and device metadata are updated accordingly and the new fstat of the file is returned to the client. For a read directory operation, we return the fstat of all the files/directories present in the given path. A read file operation results with the whole file being downloaded from the Dropbox into the local system at a path required by the fuse.

Class diagram of the Java implementation is as below:



<<Java Class>>
GateWayMain
<<Java Class>>
CreateDirectory
CreateDirectory()
createDirectory(FileArgs):List<...
validateArgsForCreateDirectory(...
<<Java Class>>
ReadDirectory
ReadDirectory()
readDirectory(FileArgs):List...
validateArgsForReadDirect...
<<Java Class>>
UpdateDirectory
UpdateDirectory()
updateDirectory(FileArgs):Li...
validateArgsForUpdateDirec...
<<Java Class>>
DeleteDirectory
DeleteDirectory()
deleteDirectory(FileArgs):Re...
validateArgsForDeleteDirect...
<<Java Class>>
CreateFile
CreateFile()
createFile(FileArgs):List<FstatR...
validateArgsForCreateFile(FileAr...
<<Java Class>>
ReadFile
ReadFile()
readFile(FileArgs):List<Fsta...
validateArgsForReadFile(Fil...
<<Java Class>>
UpdateFile
UpdateFile()
updateFile(FileArgs):List<F...
validateArgsForUpdateFile(...
<<Java Class>>
DeleteFile
DeleteFile()
deleteFile(FileArgs,FileData...
validateArgsForDeleteFile(Fi...
<<Java Class>>
Executer<T>
run():void
<<Java Class>>
DropBoxIntegration
DropBoxIntegration()
uploadFileToDrive(FileUploadData):ResponseData
deleteFilesFromDrive(FileData):int
getFreeSpaceInCloud(String):int
<<Java Class>>
FileOperationsDAO
connObj: Connection
pst: PreparedStatement
rs: ResultSet
query: String

Fig C: Metadata Schema

C. *Cloud based data-storage*

We can store data in any cloud storage due to the architecture although we have tested and implemented it successfully on Dropbox. This project can be expanded to allow storage to hybrid cloud storages in the backend. The architecture allows addition of Cloud storage accounts on the fly and maintains its metadata so as to split and store files in separate Dropbox accounts if the cloud storage space proves to be insufficient to store user data.

IMPLEMENTATION / TECHNICAL CHALLENGES

Connection Thrashing

As our database is hosted remotely, establishing connections becomes slow and expensive. To overcome this slowness, we make use of Apache Database connection pooling.
We make the Dropbox operations such as updating free space and deleting files as non-blocking I/O operations. This makes the system more responsive as the result is returned almost instantly.

File Streaming

When a file is split before uploading, we run the code for uploads for the split on different threads. As some these operations are asynchronous, we make a reconciliation job which will run periodically and make the metadata in the database consistent with the files in the Dropbox.

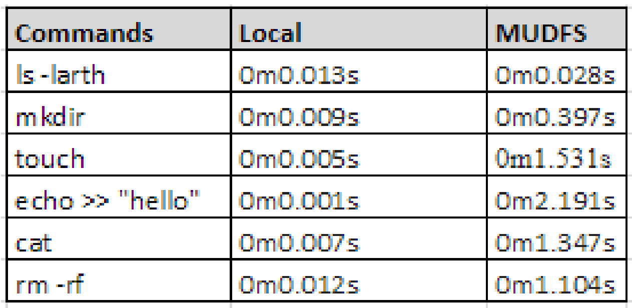
Solving the Copy Buffer Problem

Linux by default reads files according to a specific block size (4kb), therefore copying a large file does not perform well in our system as after every read the exe is called.
A possible approach to solve this is as follows -
Introduce some kind of buffer, before the exe service to call Dropbox API's is fired.

File Error Handling

Identifying and finding edge cases for file upload and download.

PERFORMANCE



Commands	Local	MUDFS
ls -larth	0m0.013s	0m0.028s
mkdir	0m0.009s	0m0.397s
touch	0m0.005s	0m1.531s
echo >> "hello"	0m0.001s	0m2.191s
cat	0m0.007s	0m1.347s
rm -rf	0m0.012s	0m1.104s

Fig D: Performance matrix (File Size = 5.2MB)

FUTURE SCOPE

Quicker updates

We plan to divide the file to be uploaded into smaller chunks and spread them across the available devices. For each split, a corresponding hash would be maintained against it in the database. When the file is updated, we need only upload the blocks for which the hash value changes. This will make the update of a file relatively inexpensive and faster.

Fault tolerance

Further, we can use consistent hashing algorithm like Karger et al. to determine which chunk should be stored in which device. We can use the concept of Raid 5 to distribute parity blocks across the servers make the system fault tolerant to failure of 1 device.

REFERENCES

[1] https://en.wikipedia.org/wiki/Filesystem_in_Userspace
[2] https://www.stavros.io/posts/python-fuse-filesystem/
[3] https://www.dropbox.com/developers/documentation/java
[4] https://aws.amazon.com/rds/