Distributed Storage System

Team member: Jingxiong Huang, Xin Lu, Wen Bo

The distributed storage system we built aims at providing redundancy as well as keeping consensus between replicas. The structure of the system is consisted of a client, a master server and three storage servers (see Figure 1).

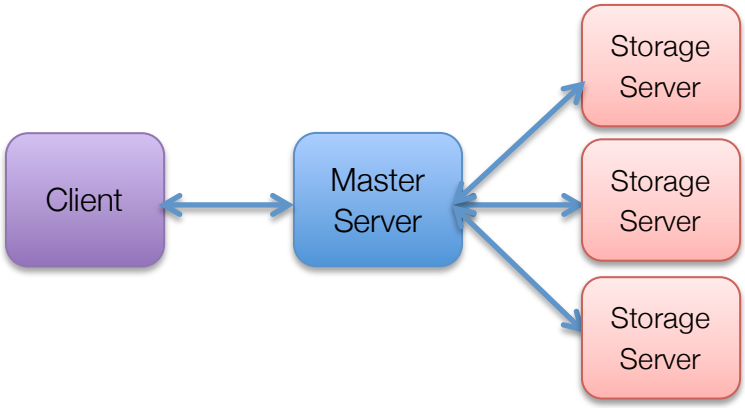

Figure 1. Distributed Storage Architecture

Storage servers are used for actual file storage on different machines. In this case (3 storage servers), the system can support redundant storage with tolerating at most one server failure. Master Server is used to manage the whole system to run properly in case of storage server failure and recovery. Client is the interface for users to use this system.

In the following text, we would mention the "Master Server Working Mechanism", "Recovery Mechanism", "Client Operation Guide" and "Test Instructions".

Master Server:

Master Server has following jobs:

1. Monitor the conditions of three separate storage server.
2. Send command to storage servers when necessary to maintain the files on them to keep the consistency.
3. Receive request from client and execute depending on the conditions of three storage servers.

The specific work mechanism of master server is as follows:

1. The master server keeps the most updated log file. By comparing its own log with storage servers' log, it will know which server is a valid server (This server has the most updated content).
2. Any command that need to modify files, e.g. upload file, remove file, make new directory, is executed using two-phase-commit protocol between master server and storage servers. In this case, we don't require 3 commit to execute the

command. Since what we implement is to keep the redundancy of the content, the master server only need to ensure at least 2 storage servers is valid.

3. When master server detects that a storage server is connected but invalid (doesn't have the most updated content), it will start a recovery process for this storage server.
4. The storage service is available only when at least 2 storage servers are valid. If there is not enough valid storage servers, the client has to wait until enough storage servers recover from a crash or fall-behind.

Recovery Mechanism:

This recovery mechanism we designed is based on log information. The reconnected storage server would execute identical commands as other good running server did and stored in their logs. Different from Raft, which only works as a naming server that doesn't do the actual storage, our system is designed for real file managing and storage. Therefore, the ability to recover a real file (e.g. image) is essential to our system. The specific operations for every command is as below:

Upload <Path> <Filename>: Request file from good running servers and store locally at the correct path

Rm <Path/Filename>: Remove local file

Mkdir <Path/Dir>: Create a directory locally at the correct path

Rmdir <Path/Dir>: Remove local directory

Mv <srcPath/Filename> <detPath>: Copy local file at srcPath to detPath and remove the local file at srcPath

Cp <srcPath/Filename> <detPath>: Copy local file at srcPath to detPath

Our recovery mechanism would track the log index of good running server and reconnecting server. The first log to run can be located by comparing the log of reconnecting server and the others. All the commands would be running under the log order one by one. While this simple model could work for most of the situations, we found cases that may cause a trouble:

37: Upload Dir1 image1.jpg

38: Rm Dir1/image1.jpg

If reconnecting server crashed before index 37 and there are two logs in this order in the good running server, the reconnecting server may meet a trouble when execute command in 37. Since image1.jpg has been deleted on other servers when executed command in 38, request to other servers for image1.jpg can't retrieve any file back. Ignoring this can be a reasonable solution since the next command in 38 would delete image1.jpg again.

However, when we replace '38: Rm Dir1/image1.jpg' with '38: Mv Dir1/image1.jpg Dir2', it can't be ignored or image1.jpg would be lost permanently on the reconnecting servers.

Our first approach to solve this problem is to transfer the file to temporary files instead of removing it permanently. Therefore, 'Rm Dir1/image1.jpg' would actually change the name of 'image1.jpg' into 'image1.jpg##tmp'. When request from reconnecting server can't find 'image1.jpg', it would retrieve 'image1.jpg##tmp' back which has the same content. However, this approach is not perfect. If we have following logs in the good running server:

37: Upload Dir1 image1.jpg

38: Rm Dir1/image1.jpg

39: Upload Dir1 image1.jpg

40: Mv Dir1/image1.jpg Dir2

and 'image1.jpg' are of different content in command 37 and 39, command 38 and command 40 would generate two files with the same name 'image1.jpg##tmp', which may confuse the later request for it. Using metadata to separate them could be a solution but a little inefficient. So we use another strategy to solve this problem. Instead of storing the file only by its name, we append "##<index>" to the filename. Therefore, 'image1.jpg' in 37 and 39 would be stored as 'image1.jpg##37' and 'image1.jpg##39' respectively, and their tmp file would be named as 'image1.jpg##37##tmp' and 'image1.jpg##39##tmp', which solved the mentioned problem.

Furthermore, keeping a large number of obsolete files is space wasting. So we designed a garbage collection mechanism to remove these temporary files when receiving command from the master server at the moment that all the servers are online and up to date.

Client Operation Guide:

We built a Linux shell-like client for users with the following commands and usage:

FUNCTION	COMMAND	USAGE
List Directory Contents	ls	ls
Change Directory	cd	cd <Path>
Create Directory	mkdir	mkdir <DirectoryName>

Remove Directory	rmdir	rmdir <DirectoryName>
Remove File	rm	rm <Filename>
Upload File	upload	upload <Path>* <Filename>**
Download File	download	download <Path> <Filename>
Exit client	exit	exit

*<Path> should be the destination directory ('.' for current directory).

**<Filename> should be a valid local file ready to submit or it would return Error info.

Test Instruction:

In our submitted document, "client", "master_server" and "storage_server" hold the executable program for these three parts respectively. To test the system, 5 terminals should be open and run the following commands in order (please use python 2 to run our program):

Terminal 1: open "master_server" run: python master_server.py

Terminal 2: open "storage server" run: python storageServer.py 0

Type "on" to connect the master_server

Terminal 3: open "storage_server" run: python storageServer.py 1

Type "on" to connect the master_server

Terminal 4: open "storage_server" run: python storageServer.py 2

Type "on" to connect the master_server

Terminal 5: open "client" run: python client.py

Common operations should run in Terminal 5 following the "Client Operation Guide". To simulate storage_server failure, you can type "off" in Terminal 2 or 3 or 4. After a few operations from client, you can type "on" in the off-line storage server to get the server to connect and recover. (The files are stored in directories "node0", "node1", "node2" under the same directory in "storage server").