SPEIT SJTU

Big Data Processing Project 4

Metadata Management Report

Lei WANG
118260910044
Tiance WU
117260910057
Yuanzhe GU
118260910034
Qingmin LIU
118260910037
Supervisor: Chentao WU

November 30, 2019

上海交大-巴黎高科卓越工程师学院
Ecole d'Ingénieurs SJTU-ParisTech

1 Problem Description

The main objective of this project is to create an metadata orgnization system. The whole system could be regarded as a Distributed File System(DFS) which could organize metadata of all files inside this system. Figure.1 shows the main structure of the to-be-done DFS system. In this system, client could send instructions to servers via basic POSIX APIs. Servers will execute the instruction and ensure that metadata stored in different servers remain the same. Different APIs should be realized such as mkdir, touch(create file), cd(change directory), ls(readdir), rm(remove file), pwd(stat), tree, etc. The metadata of the whole system could be traversed and its status could be listed. Once an instruction is sent by user, change of metadata needs to be sent to all meta data servers.

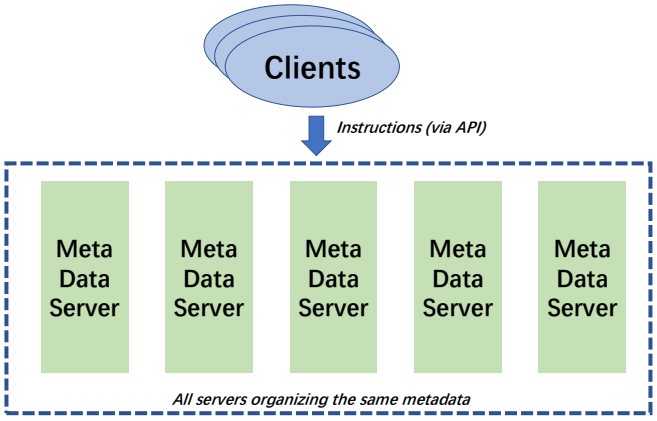

Figure 1: The metadata system to be created

2 System Design

2.1 Overall

Figure.2 shows the overall structure of our system. The metadata of files are stored in a tree where each node of tree represent a folder and each leaf of the tree represent a file. The tree of metadata is stored in every server. User could connect either of the metadata server and send instruction to

the server, the instruction will be automatically broadcasted to all other meta data servers in order that the metadata of all servers remains the same (for same instructions are sent to all servers). A user could send following instructions to modify the metadata:

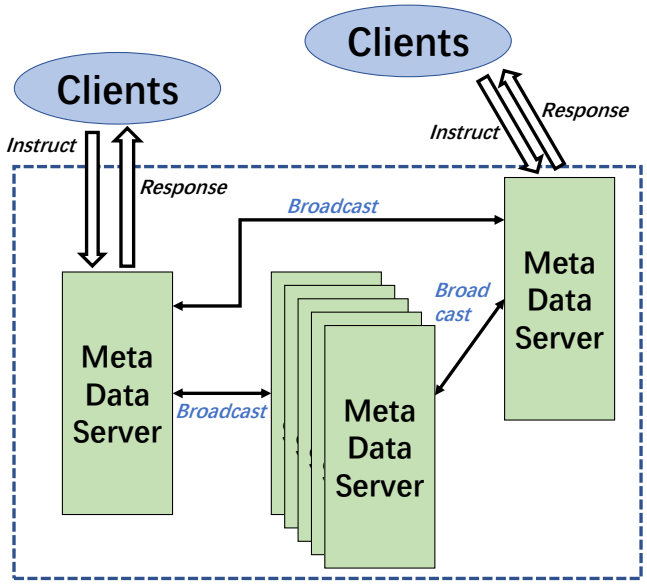

Figure 2: The system overview

- *ls* instruction to read file or folder.
- *cd* instruction to change directory.
- *rm* instruction to remove a file or a folder.
- *mkdir* instruction to create a folder.
- *touch* instruction to create a file.
- *tree* instruction to traverse the whole meta data.
- *pwd* instruction to show the present state(directory) of the metadata.

The system could hold multiple servers and multiple clients. Once a server is crushed, the user could remain connected to the file system and continue

sending instructions. In this case, other servers could remain running, except for the case that the crushed server is linked directly to user. Except for special crush cases, metadata of all servers remains the same.

2.2 Establishment of the server and clients

Figure.3 shows the main structure and functions of the server. A metadata server is holding the following values:

- A *root* treenode representing the root node of its metadata. By holding the *root* the whole tree could be listed in this server.
- A *directory* treenode representing the present state/directory of its metadata.
- An IP address representing the address of the server itself. A port number representing the port that it listens.
- A list of friends storing addresses, names of all other metadata servers co-working with the server.
- A *socket* holding the IP address and port of the server. Server will always listen to this socket.

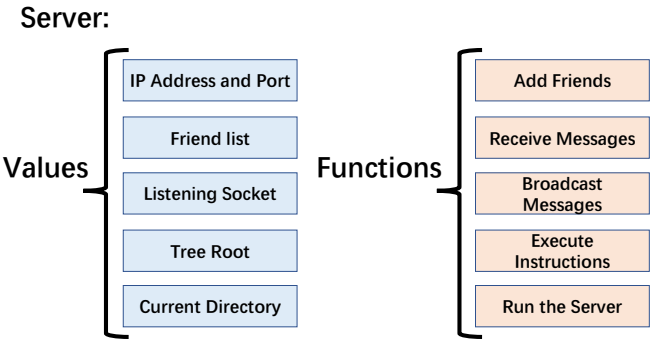

Figure 3: Values and Functions of a Metadata server

In order to use these values to run the metadata, following functions are realized in the server to complete it:

- **Add friends** and create socket to connect with friend servers;In this function, the server could record the IP address, name and port of its friend server and store them in the friend list. The friend list could be used in the following functions.

- **Receive instructions** from clients, and send the instruction to all other metadata servers co-working with the server: Figure.4 shows the basic process of this function. Data of instructions are sent by clients via *sockets*. These data are of $json$ form holding two values: *tag* and *data*. *tag* could only be two different values: *sender* and $friend$. Tag *sender* means that this data is sent by clients and needs to be brodcasted to other friend servers; tag $friend$ means that the data is broadcasted by other friend server and does not need to be re-brodcasted. *data* holds the instruction that the client send. If *tag* is *sender*, the server will extract the instruction, execute it, broadcast it to other server, and sent the execution result back to the client. If *tag* is $friend$, the server will only execute the instruction and do nothing more. Figure.5 shows a situation where one server is instructed by the client and the other server is broadcasted. We could see that the two server can execute the instruction simultaneously. If *tag* is other values, the whole instruction will be regareded as illegal.

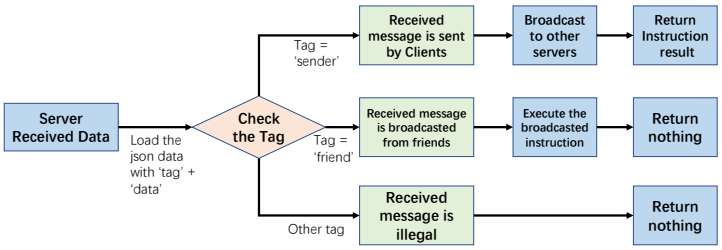

Figure 4: Process of receiving messages

- **Execute** different instructions via API: This function will check the instruction and execute it. The instruction could be ls,$mkdir$,cd,etc. This function judge the value and execute it if legal. Further expressions are explained in **??**.

- A basic *run* function to start the server: this will start the server and let it listen to the port.

Figure 5: Two servers operate simultaneously

Figure 6: Case of a crushed server on two servers(up) and on client(down)

All functions of server is written in *server.py* and functions of treenode is written in *tree.py*. *server.py* need to import *tree.py*. This file also start a server of $server1 = 127.0.0.1 : 8889$,and add a friend $server2 = 127.0.0.1 : 9999$. Another file *server2.py* will start *server*2 and add *server*1 as friend.

Functions of clients are also defined in *server.py*. The realization of client is simple, it only needs to realize the following functions:

- Connect to the server via *socket*;
- Send instructions to the server. Datas are added a $tag = sender$ and packed as $json$ form and is afterwards sent to the server. The server will load the $json$ and do operations.

Two clients are running in the file *client.py* and *client*2.*py*. *client* is connected directly to *server*1 while *client*2 connected to *server*2. If *server*2

crushed, *client*1 could remain running in *server*1 and vice versa. Figure.6 shows a situation that one server is crushed and the other server and its client remain running.

2.3 establishment of different instructions

- The *mkdir* function which create a folder. *mkdir* has only one form: *mkdir* $* **$. All other forms are regarded as illegal. The name of the folder could not include special characters such as /,<,or |. Figure.7 shows some examples of *mkdir*.

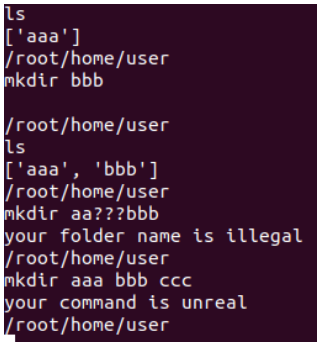

Figure 7: Samples of mkdir

- The function *tree* which shows the whole metadata. Figure.8 shows an example where aaa-ggg are folders and file1,file2,file3 are files. This function shows their relations.

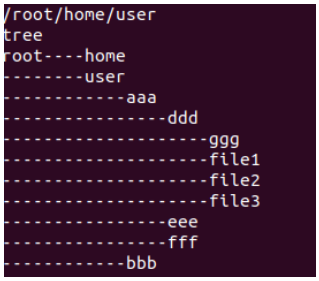

Figure 8: Sample of running *tree*

- The *ls* function which read a file or a folder. *ls* have two different cases: *ls* $* */ * */ * *$ or *ls*. All other form of *ls* are illegal. In case

ls, the function simply return all children of the present directory. In case $ls * */ * */ * *$, the function return all children of the directory $* * / * */ * *$. If that directory does not exist, the function will return that that directory does not exist. Figure.9 shows some examples of ls.

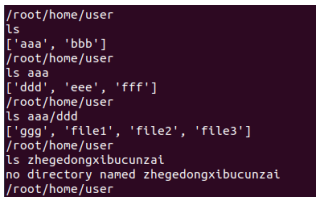

Figure 9: Different cases of case ls

- The cd function to change the directory. Firstly, the instruction cd has some spetial forms: $cd..$ will change the directory to the parent of the present directory. cd will return to the directory $root/home/user$, $cd-$ will move to the last operated directory, which is not supported for we did not build the log of instructions. A normal case is $cd * */ * */ * *$ which will move to the pointed directory if existed. All other forms will be regarded as illegal. Figure.10 shows some examples of cd.

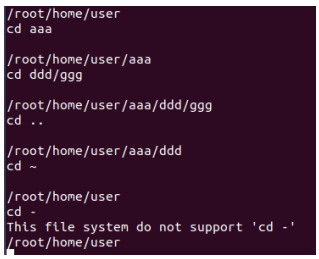

Figure 10: Different cases of cd

- The pwd functions return the current directory. Figure.11 shows the example.
- The rm function which remove a file or a folder. The rm function only supports the form $rm * *$. All other forms are regarded as illegal. Figure.12 shows some examples of rm.

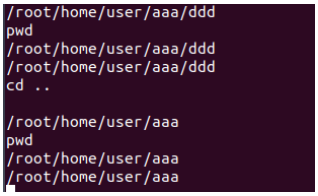

Figure 11: Samples of *pwd*

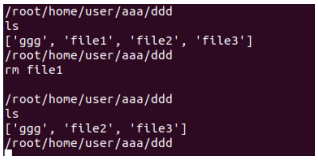

Figure 12: Samples of *rm*

- The *touch* function which create a file. It is similar with *mkdir*. Other than *mkdir*, it create a leaf of the tree instead of a treenode. Figure.**??** shows some examples of touch.

Figure 13: Samples of *touch*

3 Establishment of the System

For running the system, *python*3 is required. We could modify the IP address and port in the *main* functions in *server.py*, *server*2.*py*, *client.py*, *client*2.*py* to start different server and clients with different listening ports and their proper IP addresses. Figure.14 shows a sample where 2 servers and 2 clients are all running in a same IP with different ports. In terminals running *clients*, we could send instructions such as *mkdir*, *ls*, *rm* as we want.

Once the instruction is legal, the serever will execute it and send the result back. For every instructions (whether legal or not), the server will always send the current directory of user together with the result.

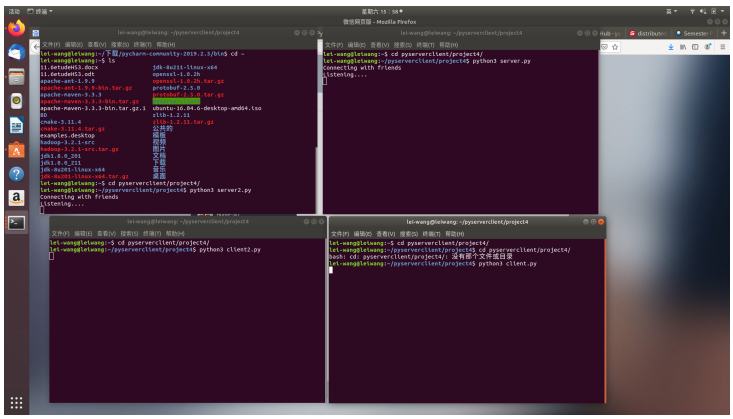

Figure 14: Two servers and two clients running together

4 Future Work

- Building up a logged file system. The logged file system could better handle cases where one server is crushed for a moment and reconnected later on. A logged file system could also support *cd−* or other type of instructions.
- Ensure Message Safety. Encode the sending data to prevent special attack to the server.
- Enable the *Copy* functions of tree for better protecting the system. Once one tree is saved in the server, it could help other servers for data recovery.
- Enable authority. This is also for a better protection of data in the file system.