Serverless DFS

System Design

The purpose of this project is to implement a distributed file system where clients can share and download files. In our design, the system consists of a tracking server and 5 clients. The tracking server keeps track of all the files in the system and their owners. And the clients can download a file, get the load of a specific peer, get a list of the owners of a specific file, and can also update their list of owned files to the server.

The tracking server is assumed to start after all clients start. In addition, the server knows the clients' addresses using a base variable for ports and the number of clients. Upon start, the server sends getList requests to all clients and stores this information in the database. These requests are sent concurrently using threads. The database is a hash table where the key is a file name and the value is a list of clients who own the file. The server has different threads that listen to clients' requests; get owners, update list requests and remove client. This design of the server side allows the server to have soft state where he only needs to store the clients' addresses. With this information, it can build the database by requesting the list of files that each client owns (client state).

On the other side, upon start, each client reads the files that exist in its corresponding home directory. After that it starts different threads to listen on 3 different ports for the following requests; getFile, getLoad and getList. getFile requests are sent from other peers to ask for a file to download. getLoad is sent from other peers to know how many other clients (peers) this client is serving. getList is sent from the server on start to build the database. Note that if a client fails in the middle, the user needs to issue update upon start.

The UI allows the user to select any of the options listed above and here is how they are handled. When the user selects updatelist, the client sends its address and a list of its files packages in a wrapper object (ownership) to the server, then the server updates the database accordingly. If the user entered find <filename>, a getFileOwners request is sent to the server, then a list of the file owners is received. If the user selected getload <peerid>, a getLoad request is sent to the peer, and an int that represents the load is returned. Finally, if the user entered download <filename>, a getFileOwners is send to the server to get the owners, if I am one of the owners, then no more work is needed. Otherwise, we send a getLoad request to the nearest owner. If the load is below the cutoff value (serving less than 3 clients) then a getfile request is sent to that peer. If its serving more than 2 other clients, we try the next nearest peer and so on. If all of them are busy serving other clients, the file is requested from the nearest owner again even if it's serving more than 2 other clients. Note that if a file is downloaded, added or deleted manually from a client's home directory, the server will not be updated unless an update command is issued. If at any point a client tried to contact a peer and the peer doesn't respond, we retry to connect 5 times. If it keeps failing, a removeClient request is sent to the server to notify it about an inactive client, then that client is removed from the database.

Based on the design described above, both the tracking server side and the client side are fault tolerant;. If either a peer client or the server is down due to a shutdown or network errors, the requesting client will retry to send the request five times. If it keeps failing, then the user will be notified using the UI.

Testing

To run tests easier, client 3 has 4 dummy threads running to simulate a client with a load of 4, which means serving 4 other clients.

The system was tested by manually running different combinations of commands. The outpt was correct for all cases except for one that will be discussed below Here is some examples of the our tests.

1- Start all clients and then start the server

2- From client 4, download f3

3- The user is notified that a local copy of f3 exists

4- From client 4, download f5

5- The client will try first to contact client 3 to download (nearest owner)

6- a message will be sent that client 3 is serving more than 2 clients

7- The server will try to contact C1 (the second nearest owner)

8- The file will be sent to client 4

9- From client 4, update list

10- In the home directory of client2, create a new file (my_new_file) with text "for testing"

11- Update list from client2

12- From the server, enter d to show the database

13- Client 2 (localhost:10002) now owns my_new_file

14- Stop client 5 (the only owner of f2)

15- From client 1, download f2

16- The client will try to contact client twice and then displays the error message

17- Start client 5 and update its list

18- Now try to download f2 again

19 - Now f2 is successfully downloaded to client 1

20 - Update list from client 1

21- From the server prompt, press d to confirm all changes has been pushed

22- Stop the server

23- From client 4, enter find f1

24- Client 4 will try to get the list from the server 5 times and then displays the error message asking to to try again later

25- Now start the server

26- From client 4, send find f1 again

27- Now a list of f1 owners is displayed.

The above sequence of commands was tested as well as more combinations and they all matched the expected result. However, the logic fails under the following scenario. When a client sends a request and the request is delivered, the client waits for the reply. However, if the destination, whether it's a peer or the tracking server, fails before replying, the system fails. Therefore, we assume that destinations might fail, which we handle, but they don't fail in the middle of a request. We also assume that the database doesn't change in the middle of the request, or at least if it does the issued request is not guaranteed to take that change into consideration.

How to run:

This project was developed using Intellij IDEA. To run any of the 3 versions, follow the steps below:

1- Open the project using Intellij IDE

2- Run the following file 5 times with 5 different server ID's ranging from 1 to 5: src/edu/umn/SDFS/ClientSide/ClientMain

3- Once all the clients are up, it will print the port on which it's listening for client requests and it will also print the peers in order of latency

4- Run the following file to start a server : src/edu/umn/SDFS/ServerSide/ServerMain

5- Client's UI has a list of commands for interaction with other clients and server. The server has an option of listing the database by entering 'd'

6- The home directory of each client can be found in src/Clients/C<client-id>