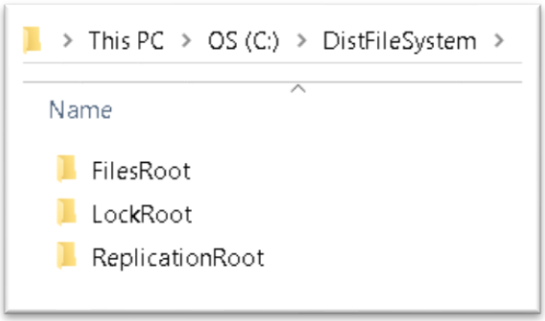
Scalable Computing – Project 3: Distributed File System

On this project I chose to implement the following 5 features:

1) File Server
2) Directory Server
3) Locking Server
4) Caching
5) Replication Server

Assumptions:

1) The servers that need disk access will work on the **'c:\\DistFileSystem\\{server_name}'** folder. When all the ecosystem is running, you'd be expected to see 3 other folders inside that one:

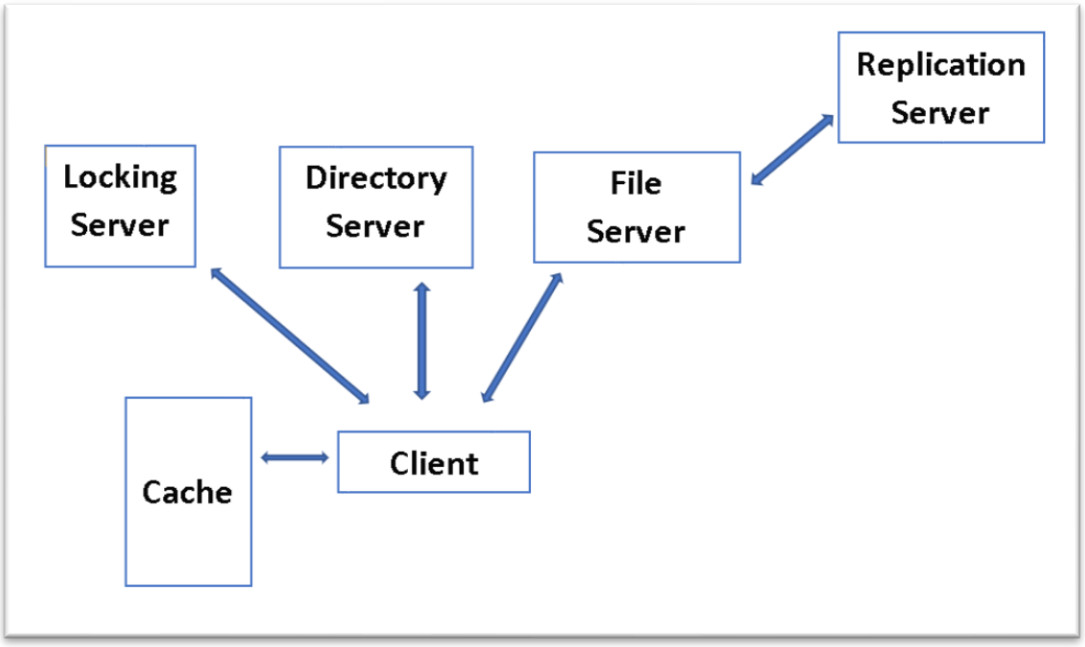


2) All files in the file server are stored in a flat file system (i.e. each file server provides a single directory in effect). Directory server maintains the mapping of full file names to server:filename mappings.
3) AFS file system was implemented.

Design decisions:

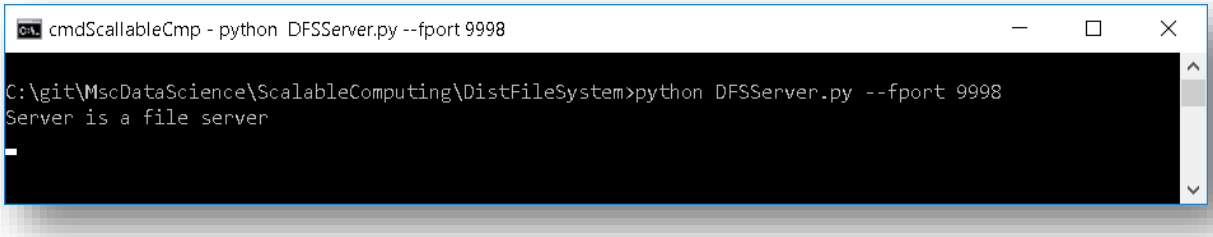
1) Client does all the communication between File, Directory and Locking Servers so those are 100% independent.
2) File Server communicates with the replication server – so client is not aware the replication server exists.
3) At the moment, there is no way to not have a replication server (a piece of the project is to build one so I didn't see the necessity to implement that functionality).
    a. However, it would be very ease to deactivate the replication tasks by changing the *replication_server_address* variable to None.
    b. That variable indicates if the file server WILL HAVE A SERVER TO REPLICATE TO
    c. For the sake of simplicity, the replication server will run on the port one number higher of the file server, so if file server is running on 9998, replication will be on 9999.
4) The replication server always have and can server the same version of the file contained on the File Server.
5) Caching is implemented on client side only.
6) Caching is clearly faster if implemented in memory but for the sake of simplicity it was implemented on the file system.
7) All servers run on "localhost"

Here is a high level diagram of the solution:



Technical Implementation:

1) The replication Server has its own file ReplicationServer.py
2) The remaining 3 services are implemented on the DFSServer.py file. Even though there are 3 services on the file, they are completely independent. When running the server, you should inform one out of the 3 parameters bellow to identify the Server role on the network:
    a. fport: File Server port.
    b. lport: Locking Server port.
    c. ds: if the server is a Directory Server
3) Example:



4) If you inform more than one, the first will be considered and the following will be ignored
5) For the sake of simplicity, the code that reads arguments from the command line can be commented and the 4 lines directly above it can be uncommented so the same Server performs 3 roles on the same port (File, Directory and Locking). This is very useful for development\debugging.
6) Caching works on the client side so there is no service for it.
7) Only supports python 3.6

The Client:

The DFSClient.py file contains the Client Class and the LocalCaching Class. An object of the caching class is instantiated on the Client constructor, so there is no need to directly interact with it.

The Client Class exposes the following methods:

- createFile:
  - creates a File on the File Server
  - Updates the directory server with the new file location
  - creates a copy of the File on the Replication Server
- openFile:
  - Asks the directory server where the file is and what it is called
  - Checks if the file exists on the Local Cache
    - If it does, asks the server for the file version (md5 hash)
    - If it is the same, uses the local copy
  - Asks the locking server if file is available
    - If not, break
  - Requests the Locking Server for a Lock on the file
  - Fetches the File from the File Server
- writeToFile:
  - Write to the local copy
- closeAndPostBackToServer:
  - posts local copy back to the server
  - adds to the local cache
  - releases the lock
- There are also methods to list the Files, Directories, Locks and Replicas

Running the client code (how to use the DFSClient class):

Below is a basic Create – Open – Write – Close workflow:

1) Instantiate a client object:
   a. This will create a local working folder for the client and initialize its cache

```
c1 = DFSClient()
```

2) Create a file:
   a. Creates a file on the server. Inform name and content.

```
c1.createFile('MyFile1', 'This is my first file')
```

3) Show Files, directories and Replicas

```
c1.showFiles()
['c5hl2qycoh']
c1.showDirectories()
{'MyFile1': ['http://localhost:9998', 'c5hl2qycoh']}
c1.showReplicatedFiles()
['c5hl2qycoh']
```

   a. We can see that:
      i. the File Server contains the internal file name

ii. the Directory Server contains a mapping from user-readable file name (MyFile1) to which server it is located and the name of that file on the server
iii. the Replication Server contains a replica of the file

4) Open File:

```
localtoken = c1.openFile('MyFile1')
Not on Local Cache -> go to server
OK
Lock on file MyFile1 granted to Client1 at 2017-12-07 20:56:52.742784
```

a. Note that:
   i. The client checks its local cache. Since it is empty, it is told to go to the server to fetch the file
   ii. The OK message comes from the locking server saying that it is ok to grant a lock to this client, thus the follow up message
   iii. The open command returns a local token. Normally this would be transparent to the user (managed by whatever file editor it is being used to edit the file), but since we are not using an editor here I have to keep it in a variable to be able to access it latter when editing or closing the file.

b. We can see the locks held by the server by running:

```
c1.showLocks()
[['MyFile1', 'Client1', '2017-12-07 20:56:52.742784']]
```

5) Write to the file:
   a. Each command writes a line:

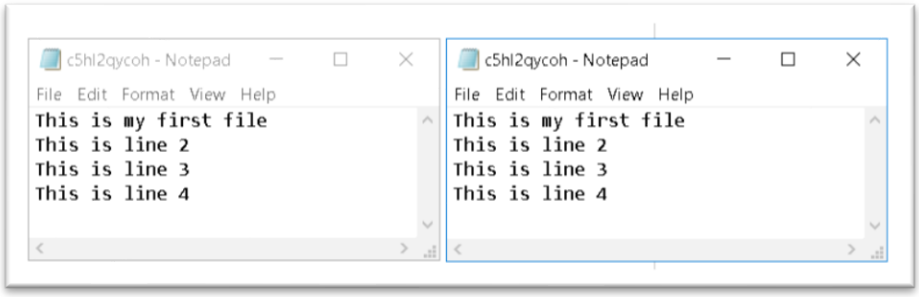
```
c1.writeToFile(localtoken, 'This is line 2')
c1.writeToFile(localtoken, 'This is line 3')
c1.writeToFile(localtoken, 'This is line 4')
```

6) Close and post it back to the server:

```
c1.closeAndPostBackToServer('MyFile1', localtoken)
File c5hl2qycoh updated- File c5hl2qycoh updated on replica server
Released Lock on file MyFile1

c1.localCaching.cache
{'c5hl2qycoh': 'ab1cfdc40a91cd60d725ceafe662089b'}
```

a. The file server tells us that the file was updated on itself and the replica
b. The Lock is released
c. The client cache is also updated with the MD5 of the file to control changes.

7) We can see that the file on the Server and the cached copy look the same:



Local caching in action:

Important: note that the "localToken" for "MyFile1" is different from the previous section of this report. This is simply because the token is randomly generted each time the file is created and I ran the code more than once to write the report. For all intents and pourposes, assume that the string **ksla007gwc** (from the second part of the report) is the same as the string **C5hl2qycoh** from the first part.

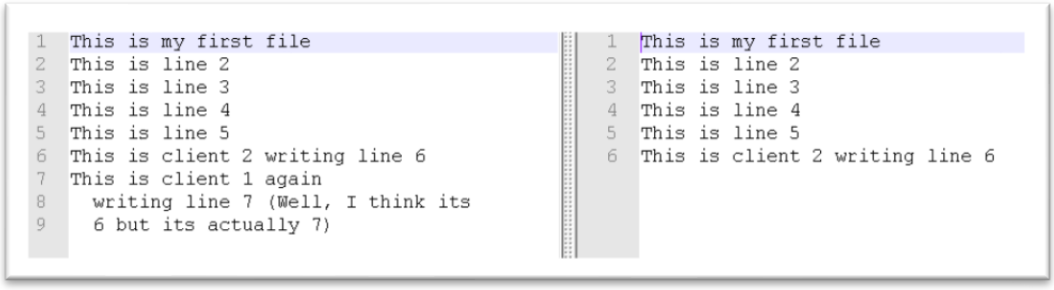
1) If client2 wants to edit the same file, it will go thought the regular process: request it from the server (assuming it isn't locked), edit and post it back.

```
c2 = DFSClient()

localtoken = c2.openFile('MyFile1')
Opening MyFile1
Not on Local Cache -> go to server
OK
Lock on file MyFile1 granted to Client2 at 2017-12-07 23:21:42.227439

c2.writeToFile(localtoken, 'This is client 2 writing line 6')

c2.closeAndPostBackToServer('MyFile1', localtoken)
File ksla007gwc updated- File ksla007gwc updated on replica server
Released Lock on file MyFile1
```

2) That action invalidates client1's cache so if client1 tries to open the same file again, its caching mechanism will:
   a. ask the server for the file version
   b. realize they are out of sync (md5's are different)
   c. Invalidate (delete) the cache
   d. Fetch a new copy of the file from the server

```
localtoken = c1.openFile('MyFile1')
Opening MyFile1
Server has a different version of the file. Request it from the server. Deleting cached
version.
OK
Lock on file MyFile1 granted to Client1 at 2017-12-07 23:21:58.216928

c1.writeToFile(localtoken, 'This is client 1 again \n  writing line 7 (Well, I
think its \n6 but its actually 7)')
c1.closeAndPostBackToServer('MyFile1', localtoken)
File ksla007gwc updated- File ksla007gwc updated on replica server
Released Lock on file MyFile1
```

3) As we can see bellow, the server has the latest version of the file (left)while client2 keeps its older version.
The most important thing here is that Client1 didn't overwrite the changes client2 made on the server just because it ahd a cached version.

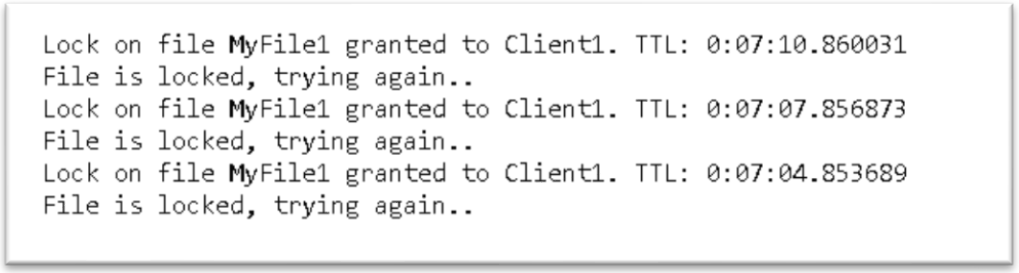


```
1  This is my first file
2  This is line 2
3  This is line 3
4  This is line 4
5  This is line 5
6  This is client 2 writing line 6
7  This is client 1 again
8    writing line 7 (Well, I think its
9    6 but its actually 7)
```

```
1  This is my first file
2  This is line 2
3  This is line 3
4  This is line 4
5  This is line 5
6  This is client 2 writing line 6
```

Further Information on the Locking Server

Each lock has a time to live (default 10 minutes) If I client ties to request a file that is locked, it will get a message saying that the file is locked and the TTL. Once the TTL expires, the lock will be released.



```
Lock on file MyFile1 granted to Client1. TTL: 0:07:10.860031
File is locked, trying again..
Lock on file MyFile1 granted to Client1. TTL: 0:07:07.856873
File is locked, trying again..
Lock on file MyFile1 granted to Client1. TTL: 0:07:04.853689
File is locked, trying again..
```