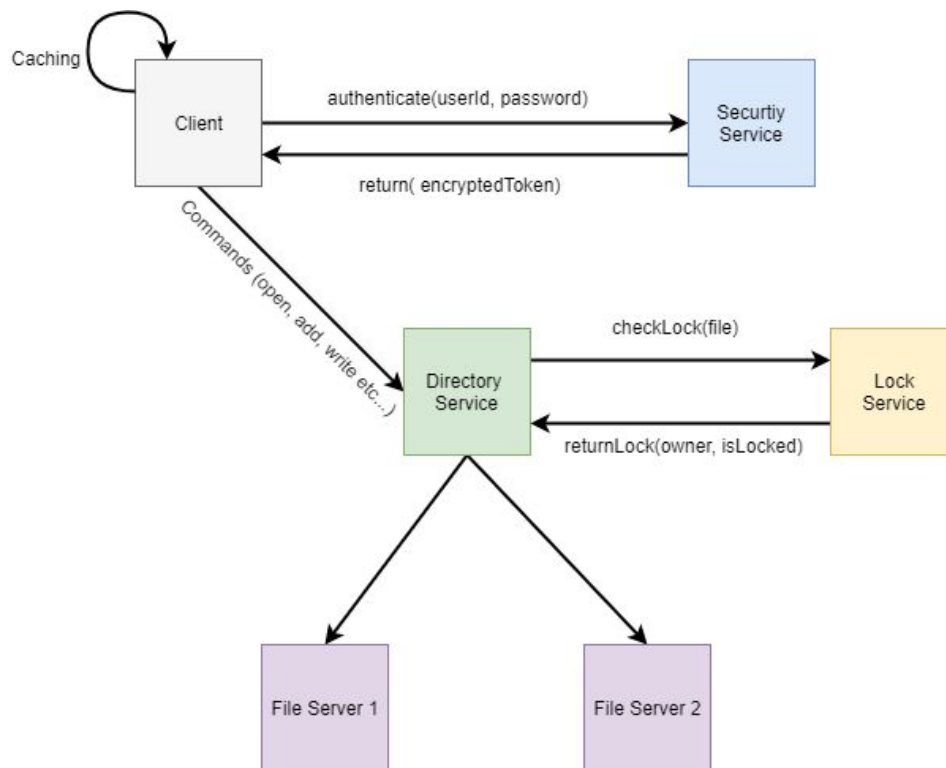


Distributed File System

Internet Applications Project



Implemented in this project

- Distributed Transparent File Access (Client)
- Security Service
- Directory Service
- Lock Service
- Replication
- Caching
- File Server

All services need to be run before any client can begin accessing data.

You can simply run the `start.sh` file or by running each service in an IDE like PyCharm.

Distributed Transparent File Access

The Client is the access point for any user to connect to the Distributed File System. From here the user logs into the system, the username and password are sent to the Security Service - which will be described later - a ticket is returned which contains an encrypted session key, which is used in all subsequent contact between the client and the directory service in order to encrypt messages passed between the clients and the servers.

The client is able to use the following commands:

- **FIND <filename>** -- This finds the server which the file exists on

```
Please Enter Command >: FIND file_1.txt  
{ "Server: ": "File Server 1", "file_path": "/files/" }
```

- **OPEN <filename>** -- Opens file. In order to commit any commands such as read or write, a user needs to open the file first, this essentially downloads the file, locks the file and stores the file locally on the clients system (caching)

```
Please Enter Command >: OPEN file_1.txt  
File successfully opened.
```

- **CLOSE <filename>** -- Closes file. Closes any file that the client has opened, this also unlocks the file for other clients to access

```
Please Enter Command >: CLOSE file_1.txt  
File successfully closed.
```

- **READ <filename>** -- Once the file is opened, a client can view the contents

```
Please Enter Command >: READ file_1.txt  
this is the content of file_1
```

- **WRITE <filename> <new content>** -- Once the file is opened, a client can write new content to the file. This is then sent back to the server to overwrite the current file stored in file server.

```
—  
Please Enter Command >: WRITE file_1.txt new content for file 1  
File successfully upated.
```

- CHECK <filename> -- Checks to see if this file is locked, and or who has locked it

```
Please Enter Command >: CHECK file_1.txt  
file_1.txt is locked by user f  
Please Enter Command >: CHECK file_2.txt  
This file is either not locked or does not exist.
```

- ADD <filename> <file server> -- Add a new file to the specified server. With Replication the main server specified will receive the primary file, all sister servers will receive a backup copy

```
Please Enter Command >: ADD new_file.txt File Server 1  
File succesfully added.
```

- LIST -- Returns a list of all the currently opened files
- HELP -- Returns a list of all the commands
- QUIT -- Self explanatory

Security Service

The security service handles the encryption of messages between the client and the servers. Upon user login, the security service creates a token comprising of a session key, server ID to which the session key belongs and a ticket which is an encrypted copy of the session key which can only be decrypted with the server key for the server to which the ticket belongs.

Accessing Security Service

```
Sending {'user_id': 'f', 'password': 'n', 'encrypted_id': 'CA==', 'server_id': 'directory_key_1'}  
{'ticket': 'RiNEIDFDKUNJCDNcIGYHXA==', 'session_key': 'NJ6ER7F10WX9Y968', 'server_id': 'directory_key_1', 'timeout': 200}
```

Encryption only occurs between clients and the servers, inter server encryption was not applied in this implementation. Contact between the client and the server occurs after the client gives the system a command, the command is processed on the client side and the data/information is sent via POST/GET over the url which matches the specific command. The json data which is sent over the url contains an encrypted version of the session key, otherwise known as the ticket as well as the encrypted version of the file data. Once the given server receives the input, it decrypts the ticket using the server key, this returns the unencrypted session key which is then used to decrypt the file and other contents.

Directory Service

This acts as an intermediary between the client and the different servers in the system (Lock, File Servers etc.). The input commands are dealt with on the the client side, these are then sent to the directory service over url via POST/GET commands. The directory service acts as a single interface for all the File Servers in the system. It loops through all file servers and returns/updates/adds files to each of the file servers which have been specified by the client. The client does not need to use the file path in any input commands, as this is handled by the directory server directly. The only time a user will need to use any indication of a file path or file server is when they intend to add a new file. This is to specify the file server which they intend to add the primary copy of this file.

The Directory Service has four main functions. The `get_directory()` function which returns the File Server containing the file, given by the `FIND <filename>` command. Like with all subsequent functions, the `get_directory()` function loops through all File Servers in the system and returns the name of the server, which contains a file with a matching name. The `open()` function simply returns json data containing the information of this file to the user. The `write()` function passes back this information to the server with the updated file content, this is then sent to the file server and stored. The `add()` function virtually does the same thing as the `write()` function although it does not require that the file exists on a server already.

Lock Service

The lock service acts as a simple semaphore for each file that is accessed by the user. The reading and writing of a file can only take place once a user has opened (or “downloaded”) the file. This is a simple requirement which makes locking the file easier. Once a user has opened the file, they may read and write to the file as much as they like. Any other user who attempts to access the information of the file will be met with an error message informing them the file has been locked. The lock is removed once the user has closed the file or quits out of the system. Any client can see who has locked the file by using the CHECK command.

Client1

```
Please Enter Command >: OPEN file_1.txt  
File successfully opened.  
Please Enter Command >: WRITE file_1.txt Update content of file 1  
File successfully upated.  
Please Enter Command >: READ file_1.txt  
Update content of file 1
```

Client2

```
Please Enter Command >: READ file_1.txt  
Error: No file of such name is opened.  
Please Enter Command >: WRITE file_1.txt new content for file 1  
Error: No file of such name is opened.  
Please Enter Command >: OPEN file_1.txt  
"Error:": "File is already locked."  
Please Enter Command >: CHECK file_1.txt  
file_1.txt is locked by user greg1
```





Replication

With the implementation of replication, once a file is added or written, subsequent “backup” files with the suffix ‘_backup’ (e.g. file_1_backup.txt) will be added to all the sister File Servers to the one containing the file. This file is then regarded as the primary copy. These backup files act only as a resource in the event a File Server goes down, they should not be used for editing, only for reading, as any update to the backup file will not affect the primary copy. One limitation of this feature is that the backup of a file is not locked when the primary copy is opened.

```
Please Enter Command >: ADD new_file.txt File Server 2
File succesfully added. _
```





Files now in the File Server 1

> 4thYear > Internet Applications > DistributedFileSystem > FileServer1 > files

Name	Änderungsdatum	Typ
 file_1	10/12/2017 18:40	TXT-Datei
 file_2	02/12/2017 15:23	TXT-Datei
 file_3	02/12/2017 15:22	TXT-Datei
 new_file_backup	10/12/2017 19:25	TXT-Datei

Files now in File Server 2

> 4thYear > Internet Applications > DistributedFileSystem > FileServer2 > files

Name	Änderungsdatum	Typ
 file_4	04/12/2017 15:08	TXT-Datei
 file_5	02/12/2017 22:30	TXT-Datei
 file_6	03/12/2017 21:54	TXT-Datei
 new_file	10/12/2017 19:25	TXT-Datei

Caching

Caching was actually implemented unintentionally. In my implementation, the system requires a client to open the file before any access/update of the file can be done, this was to make locking the file easier. When the user opens the file, rather than receiving the file path to the file on its respective server, the system responds with a json data file which contains the contents of the file. These contents (filename, file_content, server_port) are stored locally on the client side in a list (which is technically stored in memory, making access to the file and its contents quicker than accessing from the server) until the client closes the file or quits, both of which will result in the file being removed from the list. All currently cached (opened) files can be viewed by the user using the LIST command.

```
Please Enter Command >: LIST
[{'filename': 'file_2.txt', 'file_content': 'this is file 2\n', 'server_port': '8007'},
{'filename': 'new_file.txt', 'file_content': 'This is file new_file.txt', 'server_port': '8008'}]
```


File Server

The file server/s, although not a marked part of the assignment, are a vital part of a distributed file system. The file server acts as a server interface for the folder which contains the files. This system currently implements 2 file servers, although multiple file servers can easily be added. This system utilises a flat file system i.e. each file server contains only one directory which stores the files. The file servers all contain the same functions as the directory service, although the directory service as previously mentioned is an interface for each file server, whereas the file server functions apply only to its own server/directories.