Distributed File System

Introduction

This report will summarize the various components made as part of the CS7032 Distributed Systems project. The project description tasked the student to design a distributed file system which adhered 7 different design specifications. Unfortunately due to time constraints I was only able to complete 5 of the 7 components; distributed file access, directory server, security, locking and caching. A description of the design of each part will be given in this report accompanied by a command which will demonstrate the said functionality.
Brief Description

Components: Description & Testing

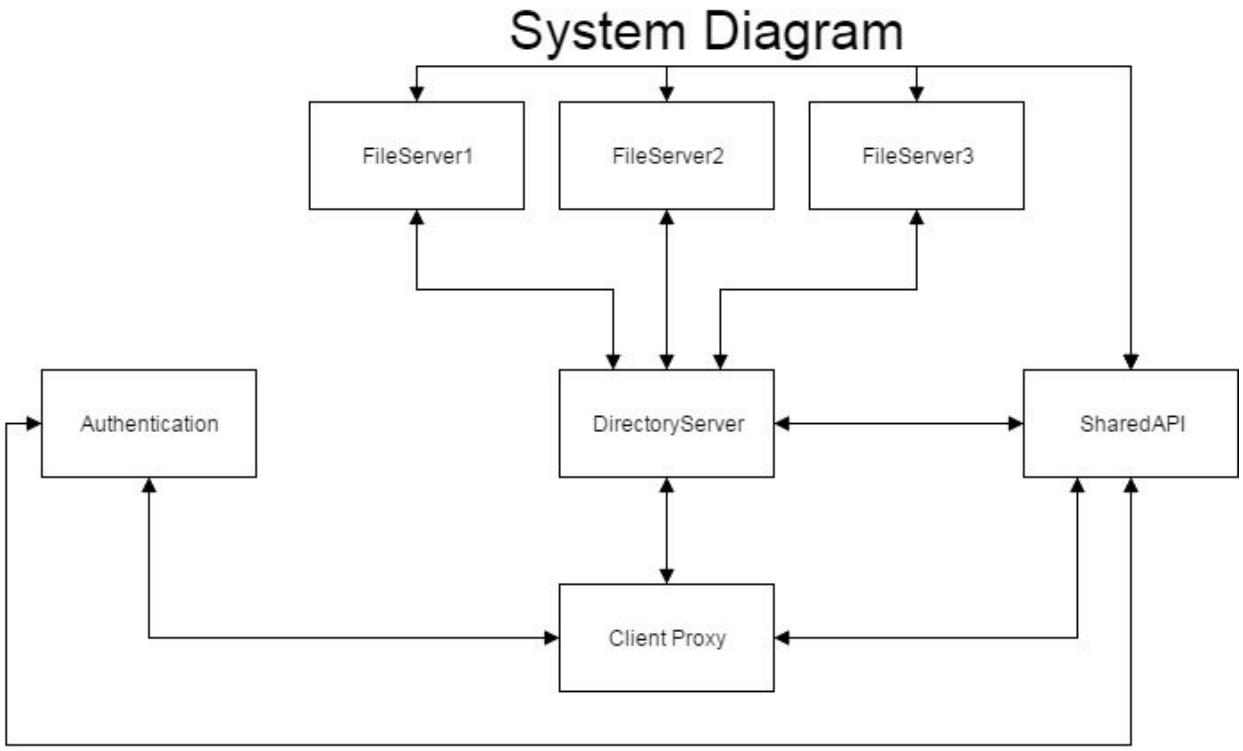


Figure 1. System Diagram

Security

Encryption XOR

This project made use XOR symmetric encryption. This means to obtain the cipher the plaintext was XORed with the appropriate key and to retain the plaintext the cipher was XORed with key. An XOR example demonstrating this symmetric property can be seen below:

$$(111011)XOR(11) = 111000$$

$$(111000)XOR(11) = 111011$$

Authentication Kerberos

Kerberos was used to authenticate users. The diagram below shows the APIs the client contacted in order to authenticate. The steps in authentication were as follows:

Step 1.

The client enters their username and password. The username entered is then XORed with the password (Client Secret). The username and client secret is then sent to the AuthInit API.

Step 2.

AuthInit receives the username and client secret supplied. It finds the password corresponding to the username and then XORs this password with the username. If the result of this operation matches the client secret the user is a valid user. The authentication server then responds to the client with a TGT (ticket granting ticket) and a randomly generate session key encrypted with the client's password.

Step 3.

The session key is obtained by decrypting using the client password and the TGT is then sent back to the authentication servers ticket granting service API. (The TGT should be stored for if the client's Ticket times out)

Step 4.

The ticket granting tickets contents is encrypted with a key only known to the authentication server (Auth key). This TGT is decrypted and the server checks if the plain text matches a string defined which I defined and if it does the client is granted a ticket. This ticket contains the clients

username, ticket timeout time and the session key. Each of these components is encrypted using the shared server key, a key known to all of the servers. Hence every command the client sends to any of the other servers is encrypted using the session key. The server which receives the command then decrypts the ticket using the shared key, validifies it (checks if the timeout time, a false ticket will not decrypt to a properly formatted time and will therefore be deemed invalid) and obtains the session key to decrypt the commands sent by the client.

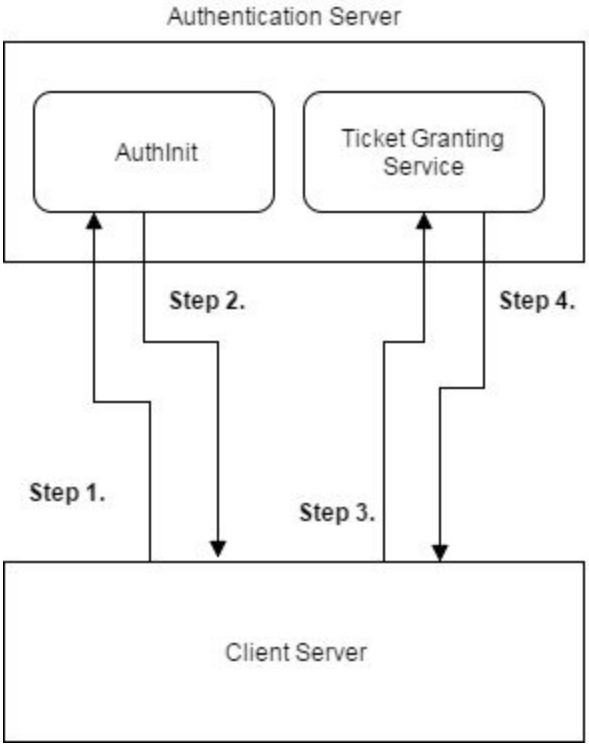


Figure 2. Authentication Server

File Server:

The files servers made were capable of performing 3 functions:

- Download
    - Directory Server sends an Instruction data type which contains an encrypted filename and ticket. The file server validifies the user and responds with a FileData data type (filename, contents & lock variable)
- Update
    - Directory server sends and Instruction_U data type containing a ticket FileData and a port number (for directory server purposes). The file server appropriately decrypts and responds with a boolean if the transaction is successful.
- List
    - Directory server sends a ticket to the list API which then responds with the FileHere data type. The FileHere datatype contains the server name and a list of the files which are stored on it.

NB: all response were encrypted using client's session key.

Directory Server:

The directory server performed 3 functions and within in these functions also managed a locking service. The functions were as follows:

- Search Files
    - The client sent the directory server and Instruction containing a file name and a ticket. The Directory server then called each files servers list API and checked whether the sought after file was within the list of files received from the file server. If found the directory server responded with a location which was just a string informing the client what server the file was on.
- List All Files
    - Similarly to search files the list all files makes use of the file server list function. It takes a ticket as an input and then calls each file servers list function and responds to the client with a list of FileHeres which the client then prints.
- Download Files
    - Download files takes an Instruction_D as an input which contains a port number, filename and ticket. It makes use of the file server download function then to obtain the relevant data. When the file is obtained by the directory server it then checks if the file is in the lock folder. If it is not in the lock folder it adds it to the lock server so if somebody else tries to download it the server will set the lock variable which accompanies the content and file name (FileData) to “L” indicating to the client to set the file permissions of the said file to read only. If the file is not in the lock folder the lock variable is set to “U” indicating its is unlocked and can be written to.
- Upload Files
    - Upload files takes an Instruction_U as an input. Hence this contains a FileData type, a port number and a ticket. The port number supplied by the client indicates where they wish to upload the file to. The lock variable within the FileData can either be set to “R” or “W”. “R” indicates to the file server that the client uploading the file only has read privileges and because of this the file cannot be removed from the lock folder. On the other hand “W” indicates the client had write privileges which means upon upload the lock is released. The client sets these variables by checking the permissions for the said file.

Client Proxy with Caching:

The client proxy allows for easy authentication and use of the directory server functionality. Initially the user must enter a username and password which is then formatted and sent to the authentication server if valid the authentication process described is completed. If the client enters an invalid username or password the program will loop back to the start.

When the authentication process is completed successfully the client is directed to the dirServ function. This function explains the commands the client can make use of and then asks the client to specify a function. The functions correspond to those made for the directory server:

- Download
    - When the user enters download it will ask for a port and filename. After these details are entered to program will first check the cache for the said file before attempting retrieve the file from the said fileserver. If the lock variable is set to “L” when the file is downloaded the files permission is set to read only.
- Upload
    - The upload function also asks the user for the file to upload and what server to upload it to. When entered the program will check the permissions the user has for that file and will set the lock variable based on this.
- List
    - Lists all the files available and there locations.
- Search
    - Searches for a file entered by the user. Searches each fileserver and returns the location of the file.
- Sign out
    - The sign out function clears the user's current directory and cache. Any changes which the user makes should be uploaded before signing out.

The block diagram below shows the flow of the client proxy.

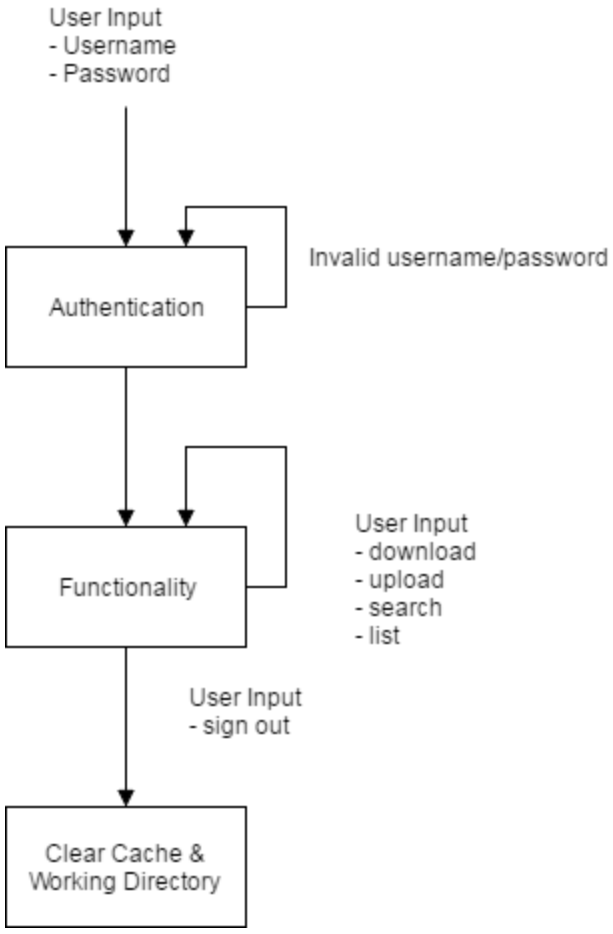


Figure 1. Client Proxy Diagram

Shared API

This library holds all of servant API definitions along with all of the data type definitions. Each server is linked to this library via their respective stack.yaml files.

Testing

Overview
The file system was designed with a git style such that you create your files and put them in a repository which the user must then upload themselves. With more time I would have added more git like functionality but just didn't have the time. I was hoping to design it such that the user could have multiple directories which they could upload to rather than just uploading from the current directory. Each file server contains 3 files: File1, File2 and File3. File server 1 also holds TF1 which is there to demonstrate locking. The file system itself has a few bugs which I would have liked to refine but it has the functionality to perform the tasks outlined in the introduction. Also within code e_variable denotes encrypted variable when the e is removed the encryption has been removed.

Server
In terminal go to the use haskell directory of the following servers.

```
cd [path_up_to_DFS]/DFS/[server_folder_name]/use-haskell
E.g.
cd [path_up_to_DFS]/DFS/Authentication/use-haskell
```

Build the project.

```
stack build
```

Run the server.

```
stack exec use-haskell-exe
```

Client
In terminal go to the Client.

```
cd [path_up_to_DFS]/DFS/Client/Client
```

Build the project.

```
stack build
```

Run the server.

```
stack exec Client-exe
```

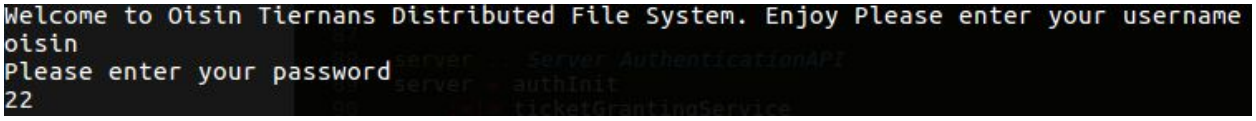
Also it may be useful to open the following directories in terminal and use the ls command after file system commands have been made:

Cache
cd [path_up_to_DFS]/DFS/Client/Client/src/Cache

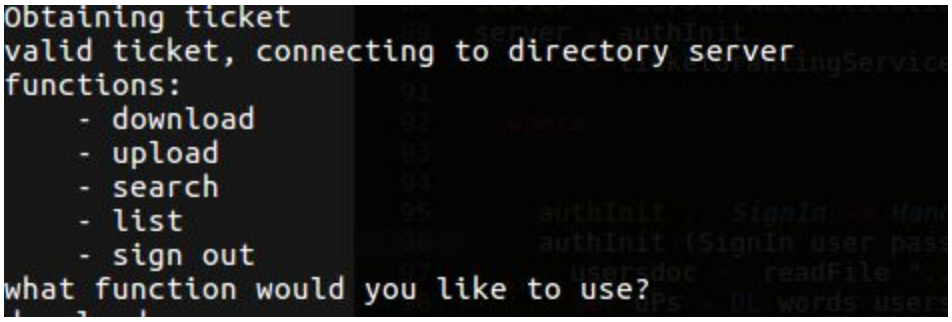
Current
cd [path_up_to_DFS]/DFS/Client/Client/src/Current

Lock
cd [path_up_to_DFS]/DFS/DirectoryServer/src/lockFiles

When the client has been run the user is prompted to enter a username and password as shown below. Enter “oisin” as username and “22” as the password.



Press enter and you will be brought to the menu. Here it will ask what function you wish to use. The following commands will demonstrate the components of the system.



Commands
Simple Download
download
File1
1

Caching
download
File1

1
The client will check its cache before requesting the file from the directory server

Locking
download
TF1
1

Unlocking
Create new text file in Current Directory and Cache called TF1
Upload
1
File will be removed from the lock directory on the directory server hence whoever downloads it next will get read permissions.