SFSA(Secure File Sharing Application)

Senior Project

System Manual

Preston Robbins

Probbins1@mail.stmarytx.edu

Spring 2022

Contents

Overview

In this document, I'll go over my senior project SFSA(Secure File Sharing Application), an application that allows users to share their files in a secure manner. The Overview goes over the problem/issue that this application addresses, the motivation behind the project, important background information about the project, my solution to the problem, and the benefits of the project. The Analysis section displays use cases for the user as well as the system and shows how the application would function. In the Design section, the tools and languages used for development, data structures, software modules, system structure, and design of the application will be discussed.

Problem/Issue

Cloud computing is a revolutionary development that allows users to access information from virtually any location. In cloud computing data is duplicated to prevent data loss. Unfortunately, this duplication of data can create consensus issues and can negatively affect a user's data. A Byzantine Fault is a possible side effect of data inconsistency on a distributed system with multiple nodes. The Byzantine Fault Problem is commonly described using the two generals analogy. In this analogy two generals(user/system) are planning to attack a fortress(task) with lieutenants(nodes) under their command. They have two options to attack or retreat. All lieutenants must reach a consensus on whether to attack or retreat. In this example one general is bad and will order his lieutenants to retreat while other will order his lieutenants to attack. This creates a consensus issue and will result in a loss for the generals. Byzantine Fault Tolerance is a distributed systems tolerance to this issue.

Traditional cloud storage also has an issue of being centralized. This means that the data of a system is concentrated in one place. A breach of this place will take down the whole system and all the data on the system will be compromised.

Motivation

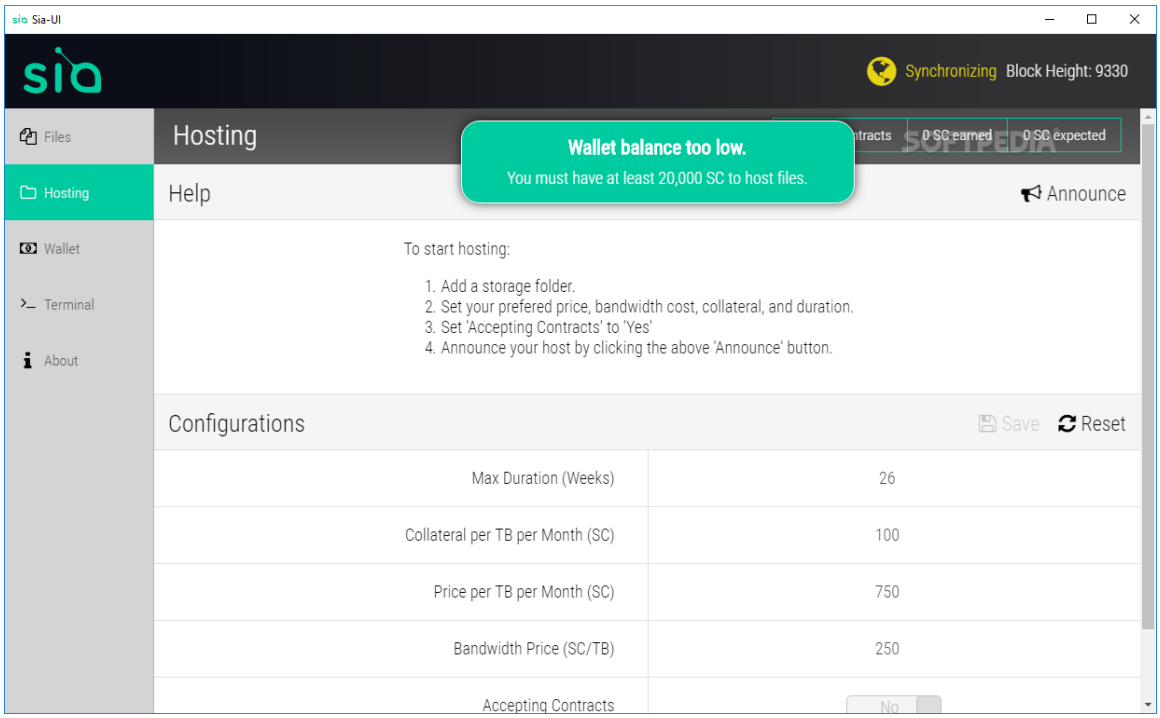
The motivation from this project comes from my previous work during an internship with Griffus Institute in the summer of 2020. During this internship I was tasked with securing sensor data on a blockchain(definition in background). For this project I used the Ethereum blockchain and a temperature sensor that was connected to a RaspberryPi. I was able to successfully send values over the blockchain however I noticed that getting live readings and updates of sensor data in real time was not practical for this type of technology. This made me want to adapt the use of blockchain for files as file retrieval does not need to be instant

Background

As mentioned previously, blockchain, is a distributed database that is shared amongst nodes of a computer network. A blockchain can be seen as a distributed ledger that is responsible for keeping track of transactions(more information in data structures). This ledger of transactions is duplicated and distributed amongst all nodes across the system. What makes blockchain secure is that it uses cryptography. Users on the blockchain have their own private and secure key as well as a public key. The private key is used to sign transactions and another user public key is used to verify the transaction.

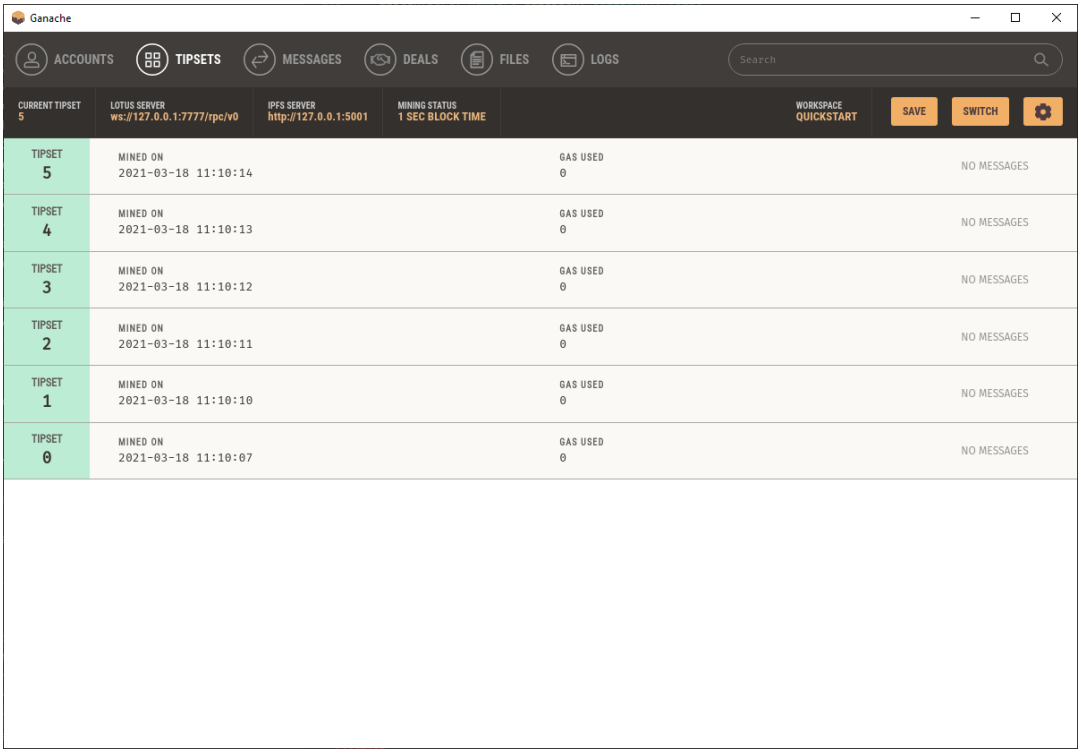
A very popular use of blockchain is the cryptocurrency Bitcoin. A cryptocurrency is digital currency that can be exchanged for goods and services. Bitcoin uses blockchain technology to verify transactions and reach a consensus . It is decentralized and operates on a peer-to-peer network. This is a popular application of blockchain using cryptocurrency. I will now mention current solutions to the problem using applications that utilize blockchain in regard to file storage.

Sia



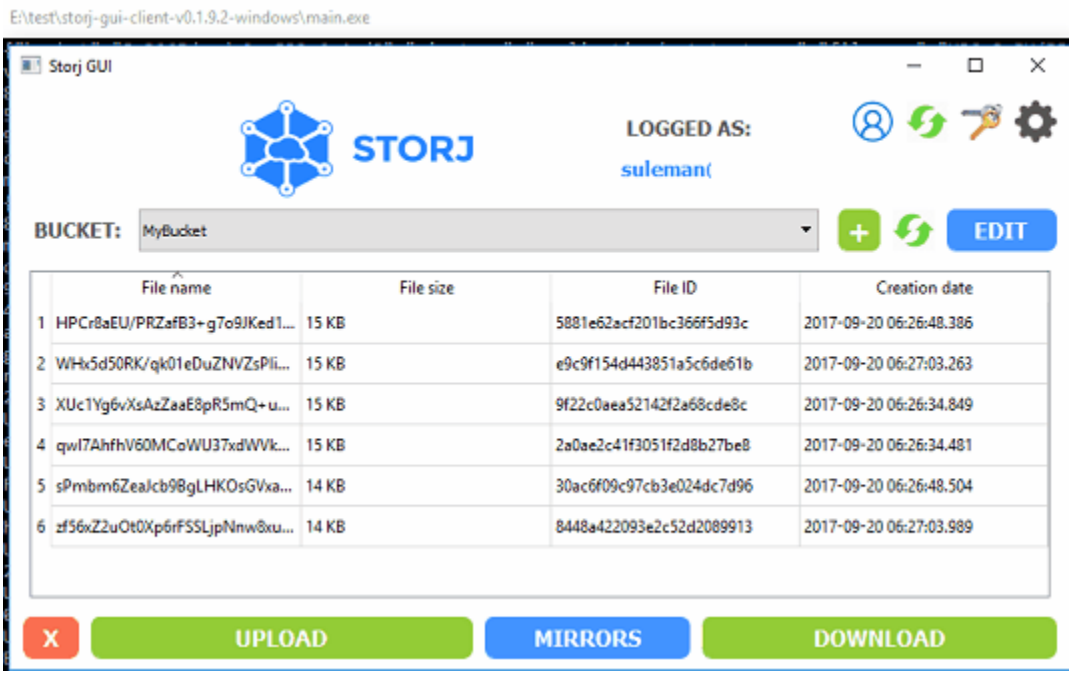
To start, I'll talk about Sia. Sia is a decentralized storage platform that uses blockchain to secure folders and files. It uses computers hard drives around the world to store files. These files are encrypted when they are stored on the system. Files are dived into 30 segments and distributed to multiple hosts across the world. If one host is compromised the attacker does not have enough information to make use of the file. This service offers high redundancy so users can download their files if hosts storing a user's data leave the system. There is not third-party management involved that can access your files as there are no central servers that are used. It is a software-as-a-service which is paid using Siacoin. Siacoin is a cryptocurrency that renters can mine and trade.

Filecoin



Like Sia, Filecoin is also a decentralized storage network. Filecoin is a distributed, peer-to-peer network. Users can sell their own personal storage on the Filecoin market to others looking to store files. Filecoin makes uses of cryptocurrency that is used to pay for the cost of transactions on the network. Instead of traditional proof of work(solving increasingly complex problems) users are rewarded based on sharing storage and contributing to deals. A deal happens when an agreement is made between a client and a miner for a storage contract. If a user wants to upload a file to the Filecoin network they can choose a list of users that are hosting their own storage space.

Storj



Storj is another example of decentralized storage. It automatically encrypts files before being uploaded. The file is then split into 80 pieces and stored amongst multiple nodes in the system. This means that there are 80 nodes that each hold one piece. Unlike Filecoin the user does not get to select the nodes that the file is stored on. Instead, nodes are selected based on reputation and local latency. Nodes that meet best suit these needs are then selected at random. This is a software as a service solution that has a free plan at 150 GB and a pro plan that cost $4/TB storage cost per month + 7$/TB bandwidth cost per month.

Solution

My solution is to create an application that utilizes the Ethereum Blockchain and its smart contracts to store and secure files. This will be a private blockchain that is meant to run and be hosted in house by a company or individual that wants to locally secure their files. The Ethereum blockchain uses a PoS(Proof of Stake) algorithm which is a protocol that will make nodes reach a consensus based on how much stake(currency) they have in the blockchain. My system will secure and encrypt files and take their hash. To share files with different users amongst the network, my system will distribute files based on users who have the appropriate public key to access the file. The hash will then be stored on the blockchain. The hash is immutable, meaning it cannot be changed. The hash can then be retrieved from the blockchain and used to retrieve and download the file in a distributed file system across nodes connected to the network. Files on the distributed file system are also immutable. A user will be able to access their files by interacting with a GUI to retrieve their file.

Benefits

The groups that would primarily benefit from this application would be anyone who needs their files secured. I would say the target audience for this application are people in the medical field. Data integrity is essential in this field because HIPPA compliance is a requirement by all practices. Another group that could use this technology is schools for storing transcripts and student information. This is important because the school needs to be in FIPPA compliance.

The on-market solutions for blockchain storage focus on global storage and store data on nodes that the user is unfamiliar with. These solutions do use encryption and split data amongst multiple nodes to combat this. Despite this I believe its best to keep documents that need to be legally protected secured on a local private network. Not all solutions on the market have a GUI or user interface. This makes it so that there is a higher learner curve for the user. My application will be made so it takes a user little technical training to store, share, and secure files.

ANALYSIS

This section covers the analysis and thought process that went into developing SFSA. It covers the hardware and software required for both developers and users of SFSA. It also covers SFSAs functionalities.

Analysis

My analysis for SFSA began by asking what a file sharing application needs and what features would be helpful when sharing files. Another important factor was considering the security aspect of the application and what the system needs to handle. The two categories I needed to distinguish were the systems functionalities and the user functionalities. Some questions I asked myself where what does the user need to worry about when it comes to file management? What can the system handle that will make sharing files easier for the user? What is the security aspect around securing files?

After asking these questions and using market applications such as Google Drive and Microsoft OneDrive I came to the following conclusions. (1) A user should be able to create an account and manage their files. (2) Managing files includes uploading and downloading files. (3) Users need to be able to share files amongst other users on the network. (4) Users should be able to mass share files using groups for ease of access. (5) The user should not have to worry about any technical details regarding the system. (6) The system should manage flow of the files including encryption, storage, and logins.

Requirements

The following paragraphs list the software and hardware requirements for both users and developers for SFSA.

Hardware and Software for a User

Tools:

- IPFS
- Ethereum

Languages:

- Python3
- Solidity
- Bash

Platform

- Developed for Ubuntu/Debian Systems

The list above shows all requirements to run SFSA as a user. Currently there is no dedicated executable. The user must run the source code from a IDE that supports Python 3. They also need to have IPFS 0.7.0 installed as well to run SFSA. The server must be ran before a node can connect.

Hardware and Software for a Developer

Developer Requirements:

Tools:

- IPFS
- Ethereum

Languages:

- Python3
- Solidity
- Bash

Platform

- Developed for Ubuntu/Debian Systems

The list above shows the requirements for developing for SFSA. You need to have IPFS version 0.7.0 installed on your nodes and server to test any additional functionalities or code that is added. You also need to have Ethereum installed on your server machine to store IPFS hashes. A developer also needs a working Python3 development environment with all libraries that are included in the source code. You also need to work on a Debian based system that supports the above tools and languages. SFSA was tested on

Functionalities

The following paragraphs show what a user can do in SFSA and what the system does in SFSA. A user is someone who intends to upload files and use the application from a day-to-day basis. The system is what manages SFSAs file system and is the applications backend.

A user can do the following activities:

- Create a account to use SFSA
- Login to account to access SFSAs main features
- Upload a file for private use
- Download private files
- Create groups
- Add users to groups
- Share files with groups
- Download shared files

The system is responsible for the following:

- Encrypting uploaded files
- Decrypting downloaded files
- Sending keys to nodes
- Connecting nodes to server
- Storing account information
- Sending messages from client to server and vice versa

Design

In this section, the initial design of SFSA will be discussed through charts and diagrams to explain each component of the application. In this section we will talk about the environments SFSA will be developed with, the data structures it uses, the software modules, the system structure, and example draft interface screens.

Environments

Tools:

- IPFS
- Ethereum
- GanttProject
- Microsoft Visual Studio

Languages:

- Python3
- Solidity
- Bash

Platform

- Developed for Ubuntu/Debian Systems

Starting with tools, IPFS(Interplanetary File System), will be used as our off the chain storage for blockchain. I chose this because you can set up local peer-to-peer storage. This is a distributed storage system that will split a file into smaller chunks and distribute it amongst the nodes on the network. This storage option can be connected to Ethereum and can interact with smart contracts. The next tool is Ethereum. Ethereum is the blockchain which will be used to store the files of a hash. It uses the PoS(Proof of Stake) algorithm to make a consensus which is important for computational resources. I chose this blockchain because I have used it in the past and have a basic understanding of how it works. I would

like to learn more about this tool. I am using GnattProject as a project management tool. This will help me keep track of deadlines and manage various development tasks. I am using Microsoft Visual Studio because it has plugins for both the Python3 and Solidity languages. I have also used it before, and I am familiar with setting it up.

Next, I will discuss the languages we will be using for this project. I am using Python3 because it supports the web3 library. This is a library that can interface with both IPFS and Ethereum. I also chose Python3 because I want to use the Tkinter library to make the user interface. There are also various encryption libraries that Python3 can use for our files. Solidity is a programming language that is used to program smart contracts on the Ethereum blockchain. I am using this program because it allows me to create more elaborate consensus decisions when it comes to nodes and will be the main language, we use to put our file hashes on the blockchain. The last language I will be using is bash. Bash is going to be used to setup the network and write scripts that will run and maintain the network in the Ubuntu environment.

The platform I choose for this project is Ubuntu/Debian based systems because I am familiar with the platform. While doing research most development with blockchain tends to be done on a Linux system. So far, the development tools for Windows are lacking. From previous experience I had success with this operating system.

Data Structures

```
@dataclass
class Account:
      username: string
      password: string
      privateKey: string
      publicKey: string
      keyRing: List[string]
      fileHashes:List[string]
      fileName:List[string]
```

The above image is a structure created in Python3 displaying a user Account. This structure is the one of the main data structures for this project and contains the rest of the data structures for this project which will be explained line by line below.

```
username: string
```

The first data structure is the username variable. This variable will hold the username associated with a user account. The username will only be known by the user. This information will be encapsulated.

```
password: string
```

The second data structure is the password variable. This variable will hold the password associated with the user. The password will only be known by the user and will be encapsulated.

```
privateKey: string
```

The third data structure is the variable privateKey. This variable will hold the users private key. This will be something that will be encapsulated by the system. The average user will not need to know their own private key to use the application as the system handles the signing and encryption of files.

```
publicKey: string
```

The fourth data structure is the variable publicKey. This variable will hold the users public key. This will be public knowledge and does not have to be a private variable. The user will not need to worry about their public keys as the system will handle decryption of files.

```
keyRing: List[string]
```

The key ring will hold public keys shared by other users. This will help the user access files that are encrypted and can only be decrypted with a user's shared public key. The system will handle this part.

```
fileHashes:List[string]
```

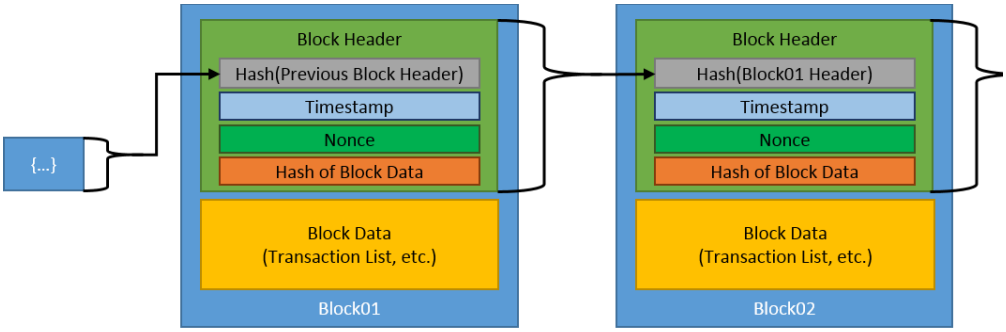
The array fileHashes will hold a list of file hashes with each position corresponding to the filename array below.

```
fileName:List[string]
```

The array fileName will hold a list of file names that corresponding to the fileHash array.

```
John221
password22
mm2m131wewq
sdasdqweqeqqw
[sdasdqweqeqq,
asdaasd11qaaW]
[xzczxa123,qweqwe221]
[Medical Documents,
Financial Documents]
```

The above is a sample instance of the Account structure. Each example in the instance corresponds to the order the structure is presented in.



The above image is a example of what a blockchain data structure looks like its definitions what each part in the block will be shown below.

Block Header:

This is used to identify a particular block on the entire blockchain.

Hash(Previous):

This contains the hash of the previous block header.

Timestamp:

This contains information about when the block was created.

Nonce:

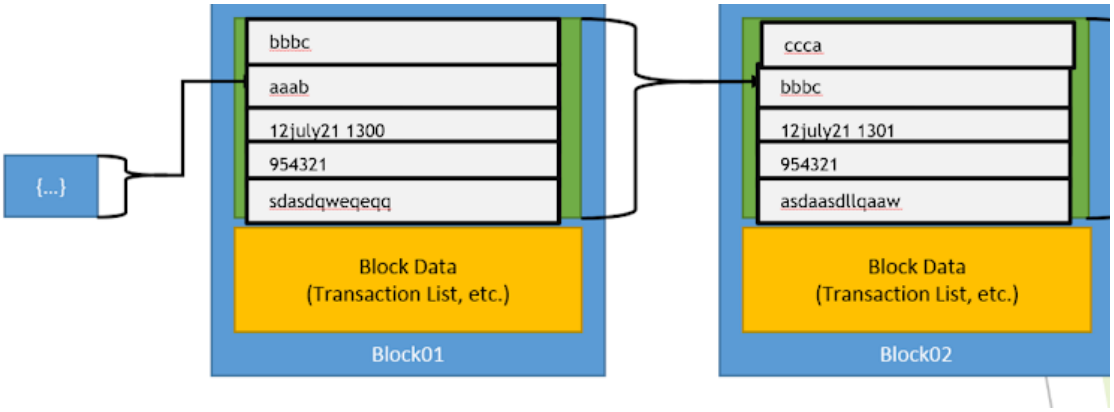
This is the variable that is incremented by proof of work.

Hash of Block Data:

This is where we will store our data as a hash on the blockchain.

Block Data:

This is a list of transactions that were confirmed in the block.



The image above is what the data structure would look like if it had information in it while running on the blockchain.

Software Modules

Account Management/Login:

This is a module that will let the user create a account and login to access all the features of a system.

Upload Files:

This module will let the user upload files to the system. The user needs to be able to interact with their local directory and the file needs to be stored on the system.

Download Files:

This module will let the user see what files are available to them and will let the user download the file from the system and on their computer.

Group Manager:

This module will let the user create and manage groups that they will share files with.

File Sharing Manager:

This module will let the user manage their own files they uploaded to the system and see who they shared it with.

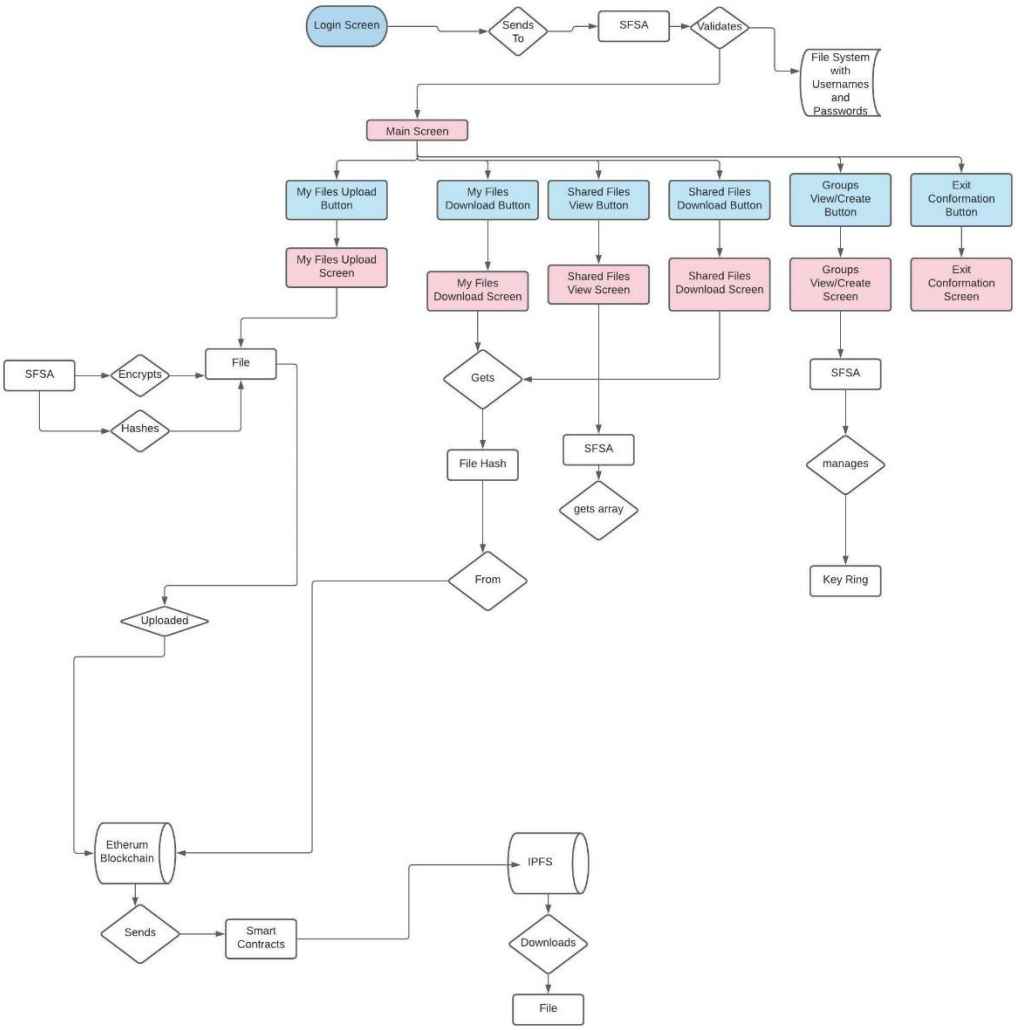
Private/Public Key Manager:

This module is a system module that will manage public and private keys of users on the system. It needs to be done in a secure way so users are able to access files on their system.

Hashing Module:

This module is a system module that will hash the files and upload them to the block chain.

System Structure



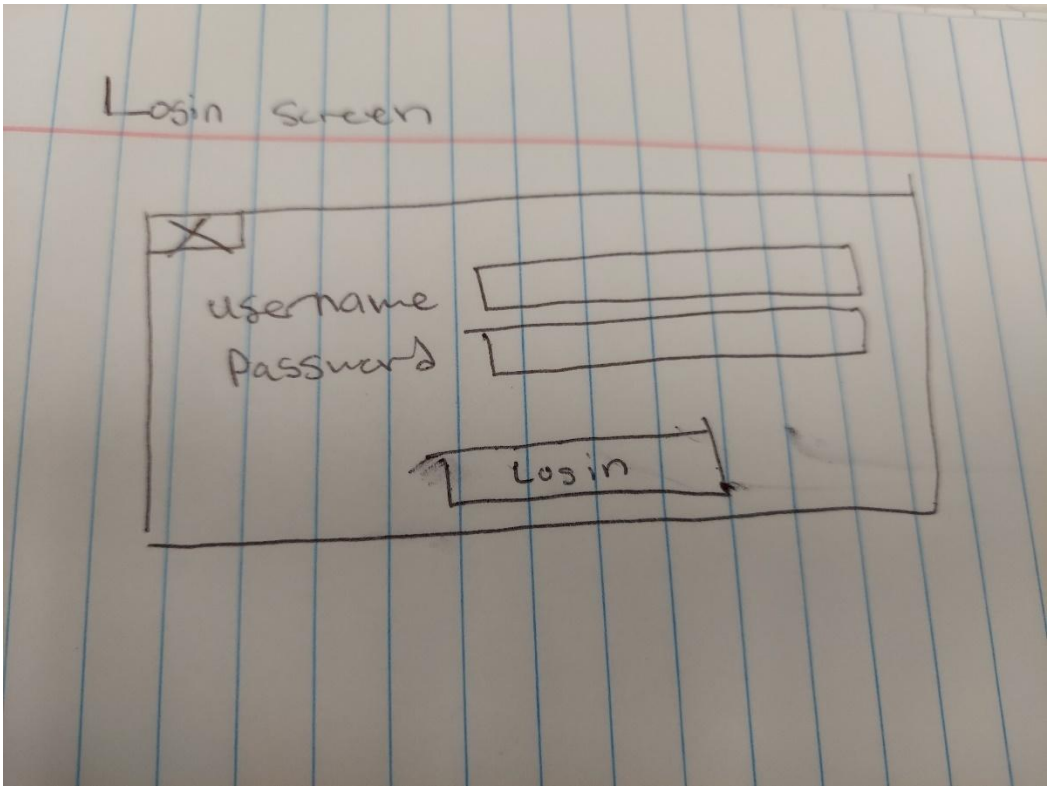
The above screen shows the flow of control for the system. It starts at the login screen and the system then confirms the login with a File System that contains the usernames and passwords is shared with all nodes via FTP. The user then chooses the main screens which will interface with the system. The files are encrypted with a user's private key and a hashing algorithm is used for the file. The file hash is then stored on the Ethereum blockchain. The blockchain then activates a smart contract which will store the file on IPFS. Depending on the option the user can either upload or download the file. The Groups screen will show the user what groups they shared a file with. The system will update the

structure that contains public keys for an account. The shared files screen will allow a user to view what files they shared and what files are shared to them. The system will look through a array of files associated with the user and check to see which ones are shared.

Interfaces

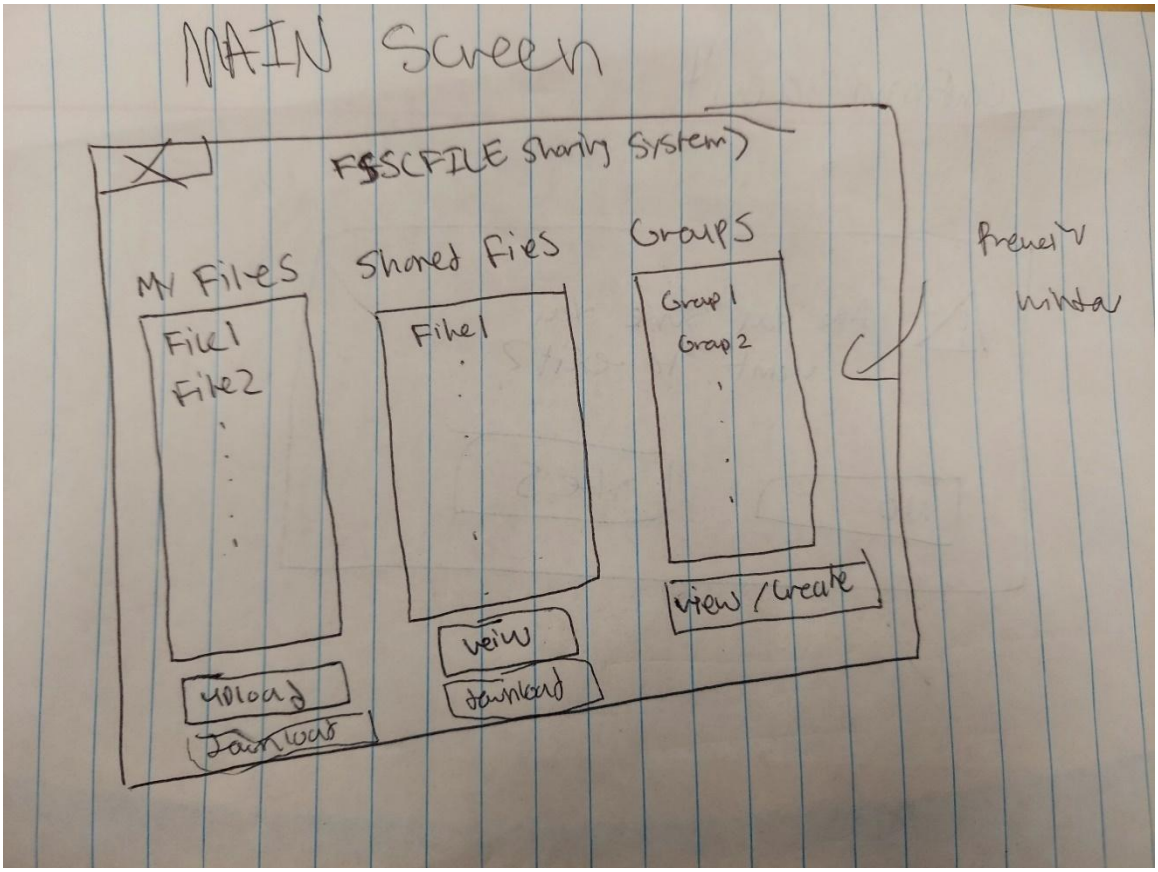
This section will show what a user can expect to see when opening the application and using the system. The following images are sketches of what features the user interface will have.

Login Screen:



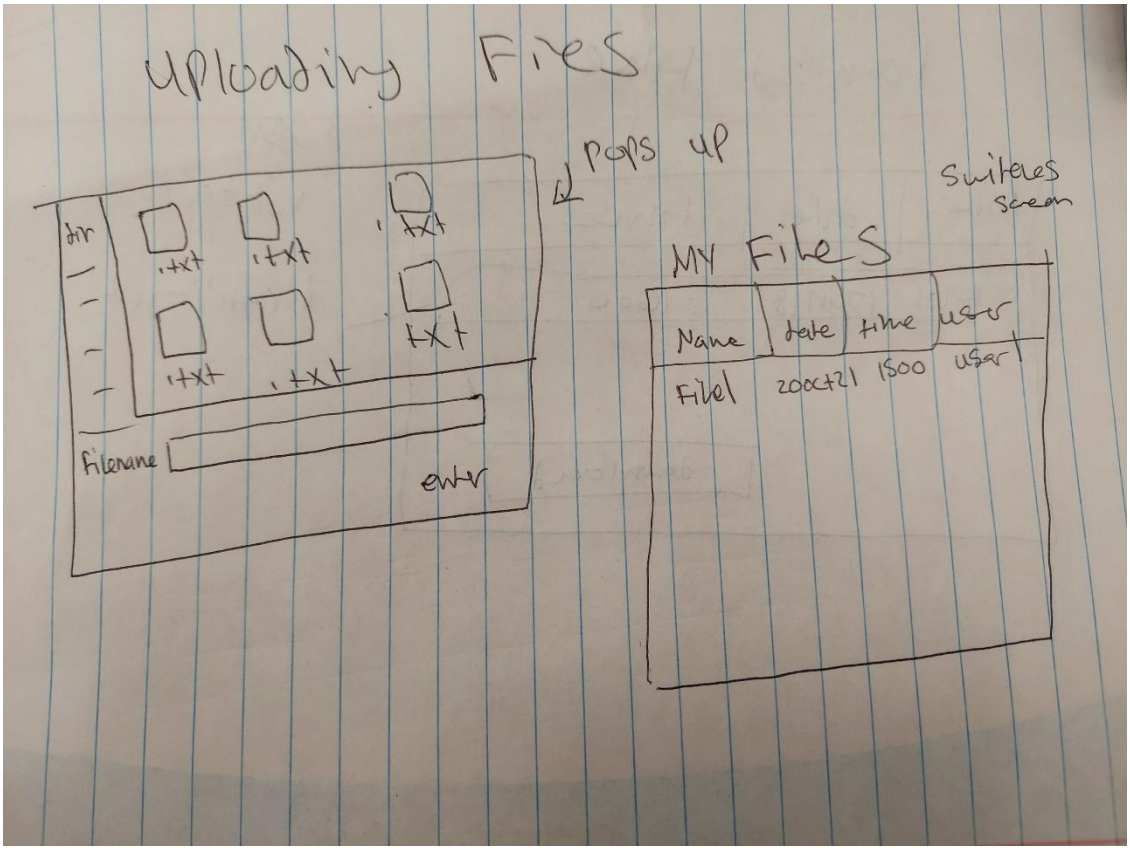
The first screen the user will see is the login screen. Upon clicking the executable for the application, the user will be asked for a username and password. They then have the option to login or click a red X to quit the application.

Main Screen:



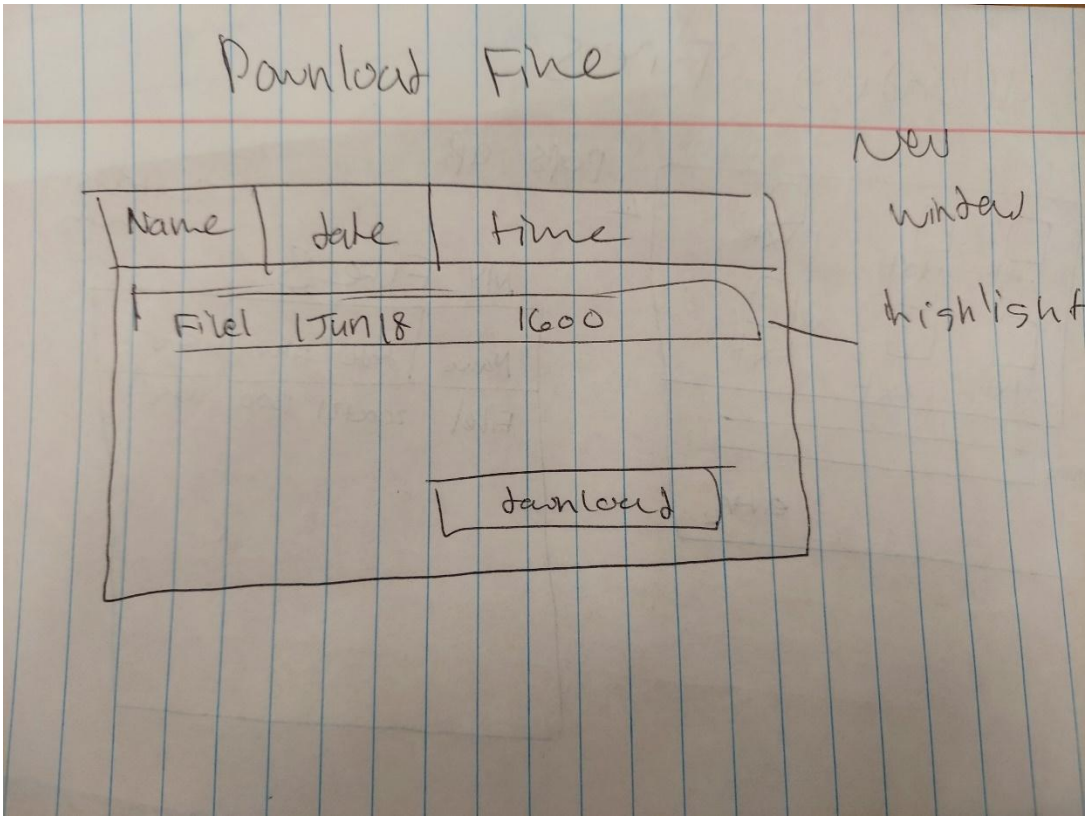
After logging in the user will view the main screen of the application which will display the three main categories My Files, Shared Files, and Groups. Each of the categories has a preview window that shows recent additions. Below the preview window are interactive buttons. Below the My Files preview window there is a option to upload files from a user's computer, or download files from the application. Under the Shared Files the options to view the files or download the files are present. Under Groups preview window the user is presented with the option to view or create groups to share files with.

Uploading Files:



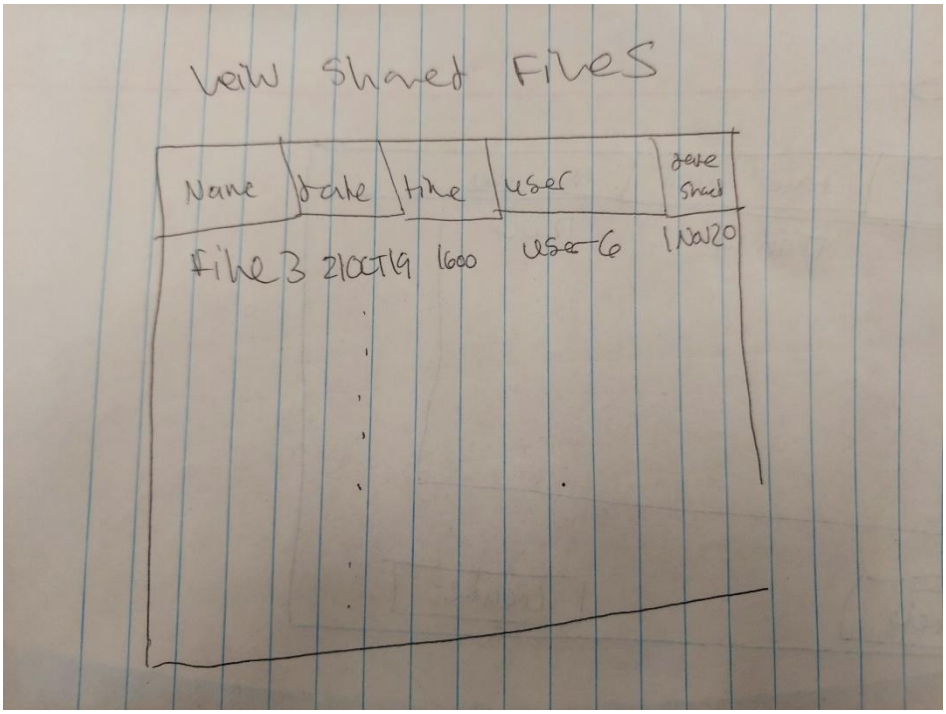
After clicking on the upload button under the My Files Screen the user will be greeted with a pop-up window of the users directory. In this directory they can browse their hard drive for files to upload. On the right hand side is a preview window of files already uploaded. The My Files window will show the name of the file, the date the file was uploaded, the time it was uploaded, and the user who uploaded it.

Download File:



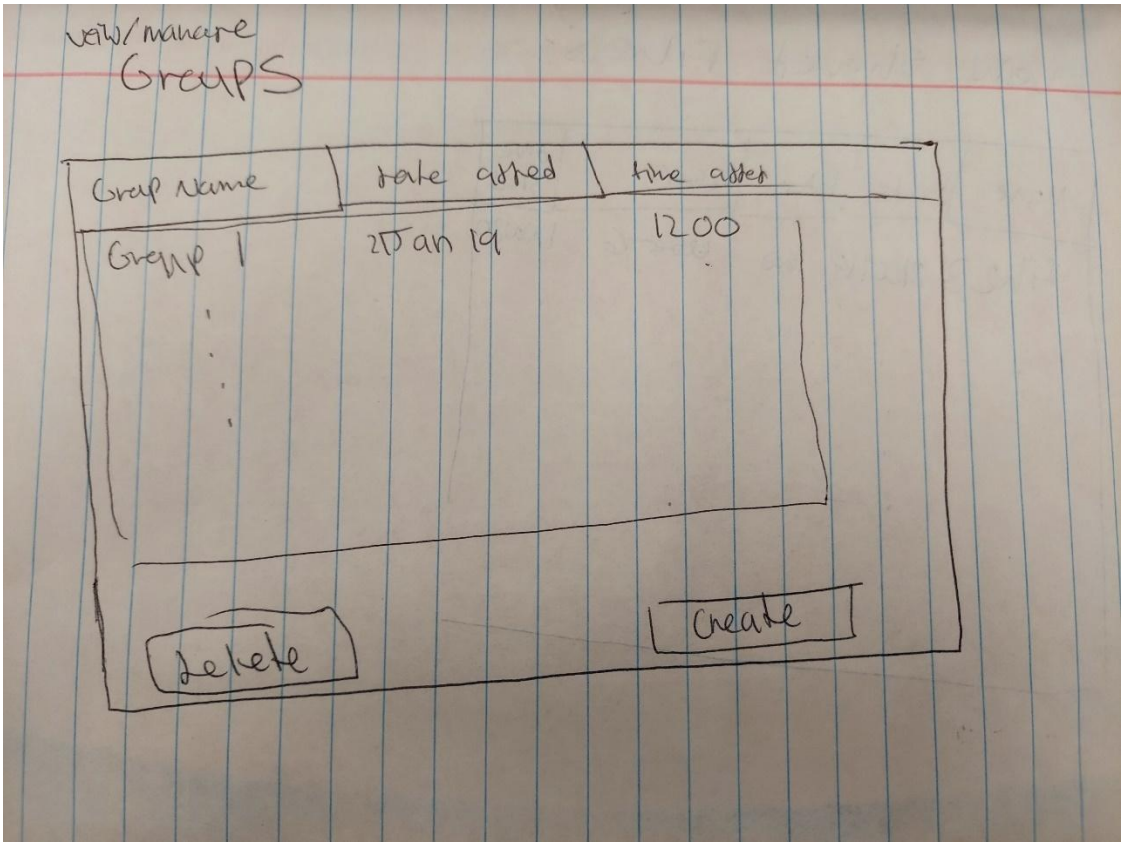
After the user clicks the download button under the My Files preview window a pop-up window will display. This window has a list of files available to download. The list will show the file name, the date it was uploaded, and the time it was uploaded.

View Shared Files:



After the user clicks the view button under the shared files screen, they will be greeted with a pop up window. The view screen shows files that the user has shared and what other files users shared with them. The popup window will contain a list of files with the file name, date the file was uploaded, time the file was uploaded, the user who uploaded it, and the date the file was shared.

View/Create Groups:



Upon clicking the view/create groups button under the Group preview window the user will be shown a pop-up window as shown above. The popup window will display the group name, date the user was added to the group, and the time added to the group. There are also delete and create options for the group that the user can input.

INSTALLATION

This section goes over the installation of SFSA for both developers and users. It includes samples screen shots for how to install the program.

Developer Tools and Source Code Files

The tools required from development are IPFS, Ethereum, and Python3. I would recommend installing Ubuntu 20.0.4 as well. Use the following links to setup your environment for development:

IPFS: https://docs.ipfs.io/install/

Ethereum: https://geth.ethereum.org/docs/install-and-build/installing-geth#install-on-ubuntu-via-ppas

Python3: https://phoenixnap.com/kb/how-to-install-python-3-ubuntu

For the application to work you must have IPFS 0.7.0 installed. You will need to manually rollback your IPFS version using IPFS update command. You can read the man page on IPFS for more info. I recommend setting up a virtual machine environment that runs Ubuntu. It is recommended to test any new changes with at least 3 machines. One machine will have the server application and the other two are node applications. You can use any preferred method of setting this environment up.

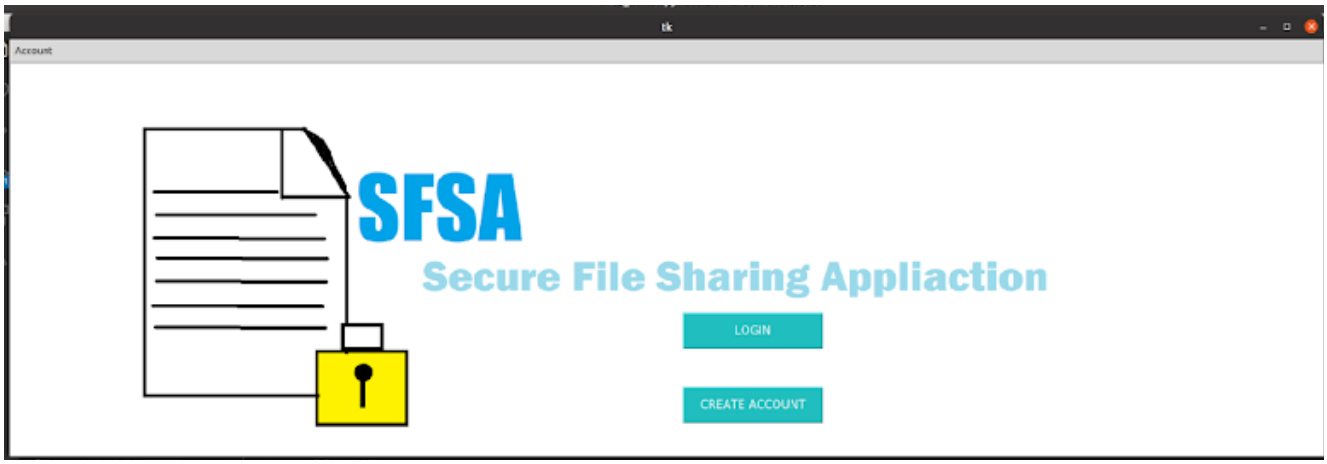
Installing for a User

A user needs to have IPFS version 0.7.0 and can use the link in developer tools to access the installation info. Like a developer the user must rollback to version 0.7.0. To get the files a potential user must email me, and I will send them the source code. As of now there is no executable for SFSA. The user needs to install python3 and all libraries imported in the source code for it to work. They also need to connect to a server that should be setup by a admin. Once the user has the files, they need to open them in a environment that supports Python3 and run SFSA_Runner.py.

Sample Sessions

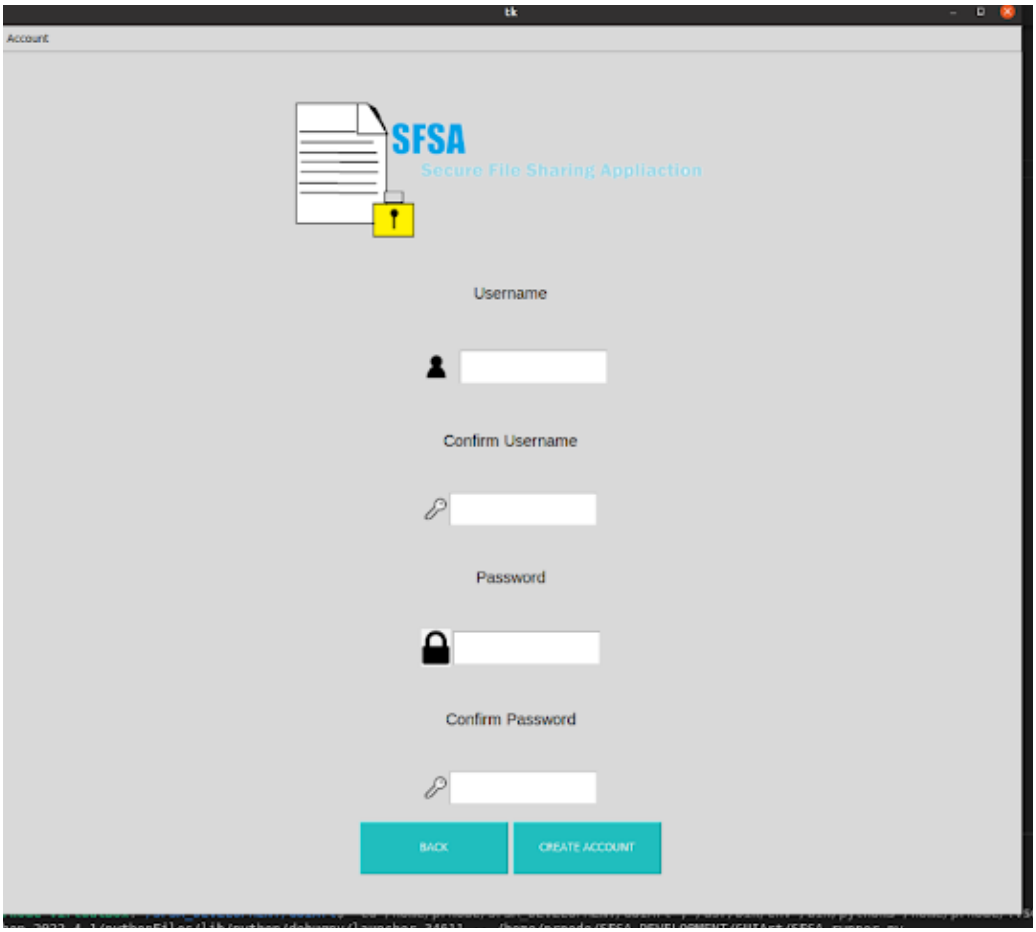
This section goes over what a sample session in SFSA will look like for the user. It lists all current functionalities that a user can do with this current build. In this sample session we create a account called UserAccount and share a file with another account Superpositive01. We go through all the main features of SFSA.

Welcome Screen

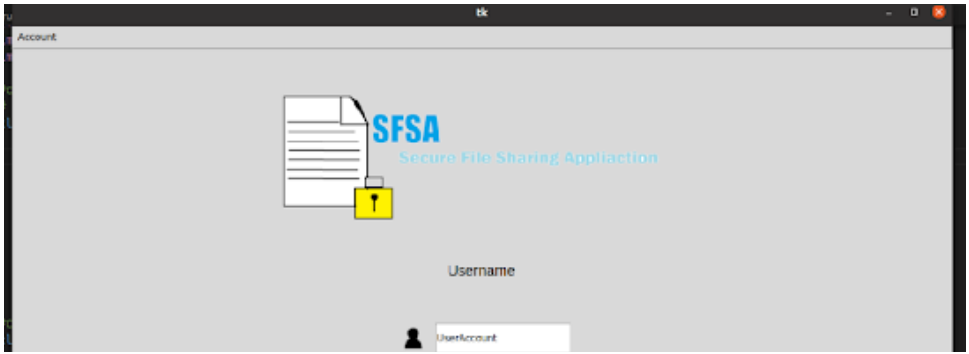


Once The user runs the application they are welcomed by the above screen. The user has two options to Login or Create a Account.

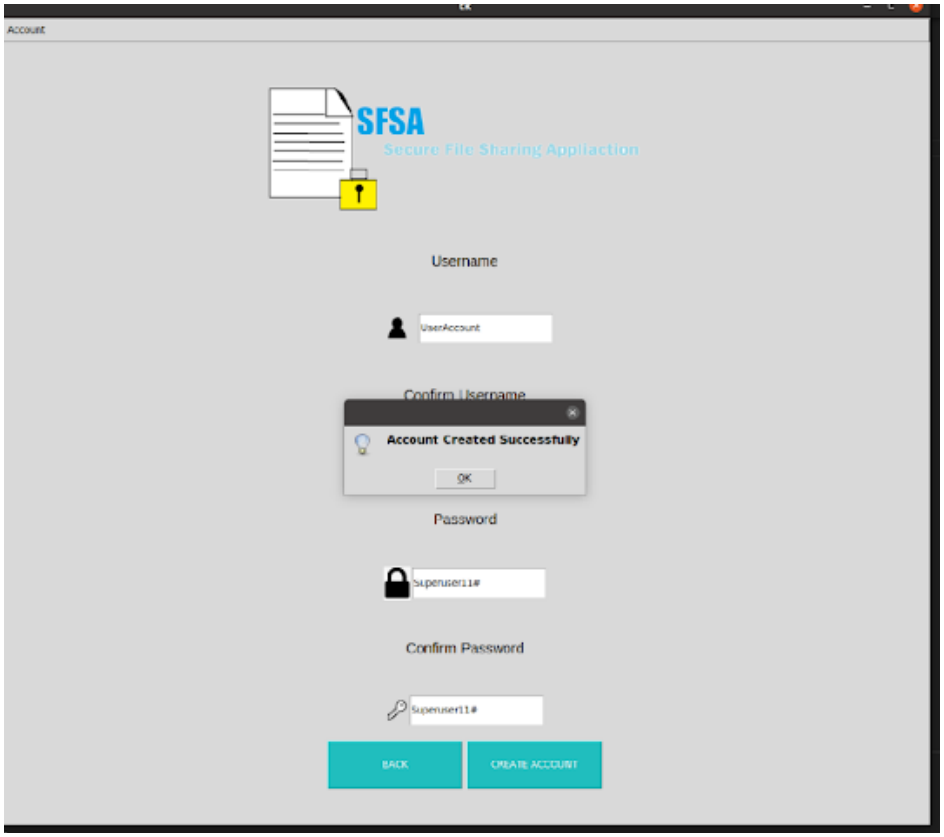
Creating An Account

To 

use SFSA a user must create an account. The above screen can be accessed from the Welcome Screen.



The above image shows what sample user input looks like. Refer to the Test Plan for error codes regarding this screen. A user must adhere to the username and password requirements for creating an account.

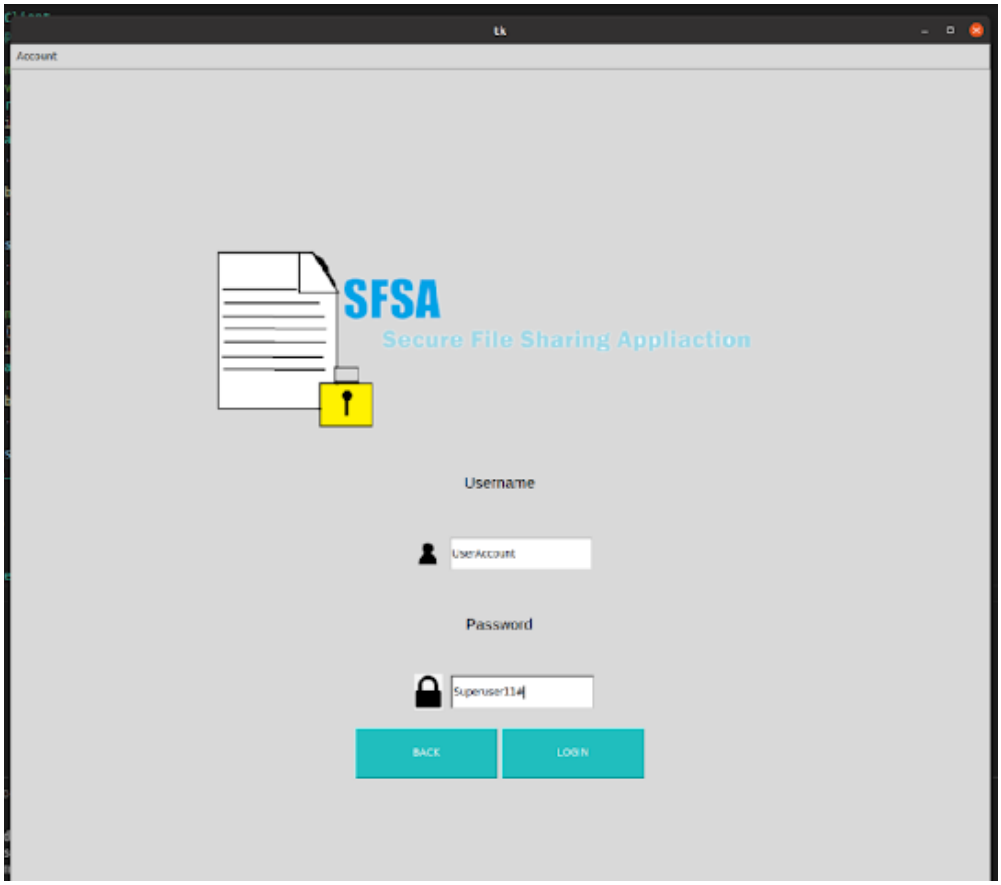


The above image shows what message you get after you successfully create an account.

Logging In


tk
Account
SFSA
Secure File Sharing Appliaction
Username
Password
BACK
LOGIN

The above image is the log in screen that can be accessed from the welcome screen.

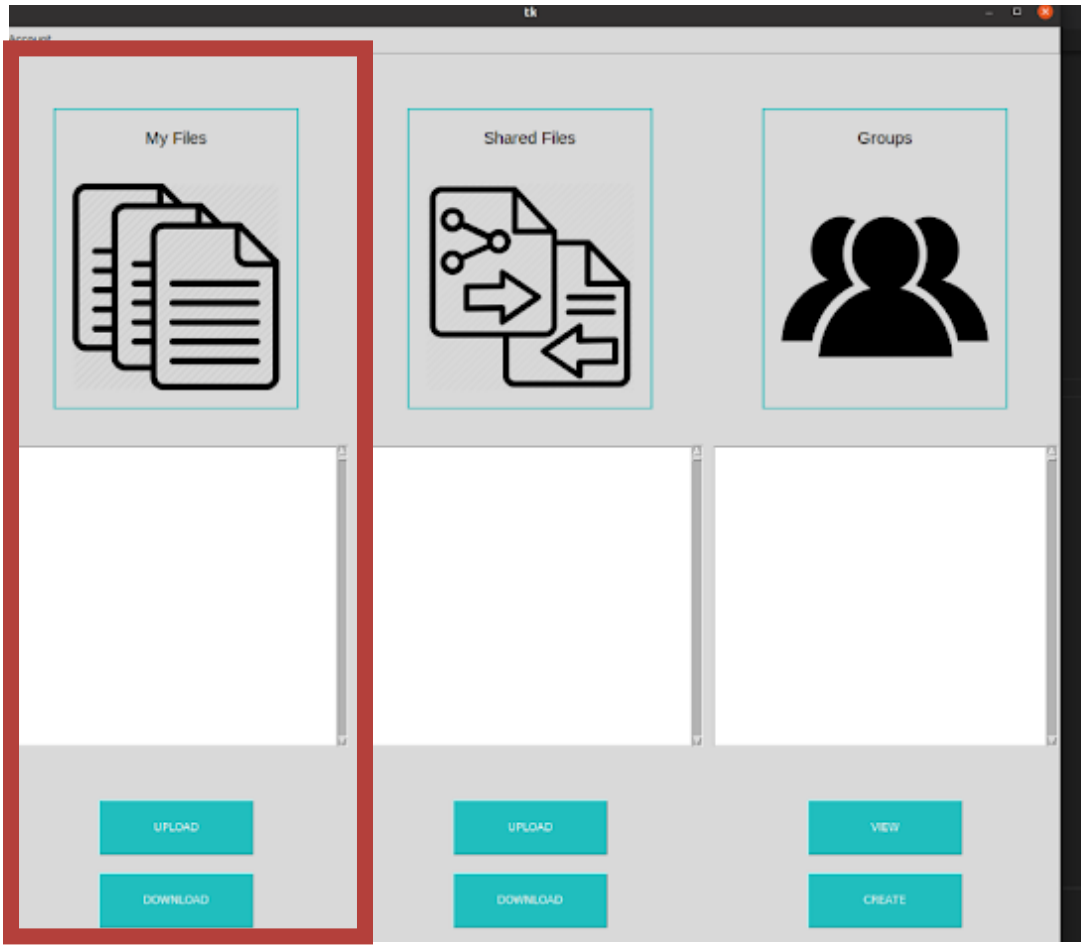


The above image is the log in screen with our account information we just entered. Please refer to the Test Plan for possible error codes.

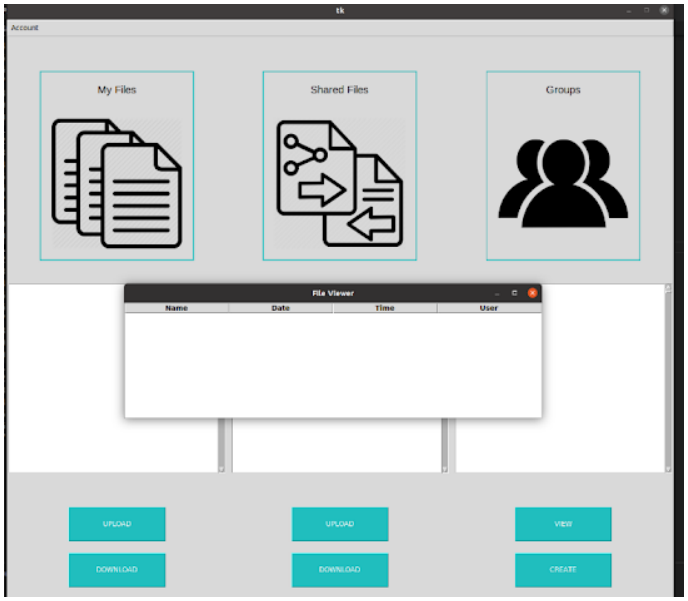
Main Screen

After a successful login the user is welcomed to the above screen.

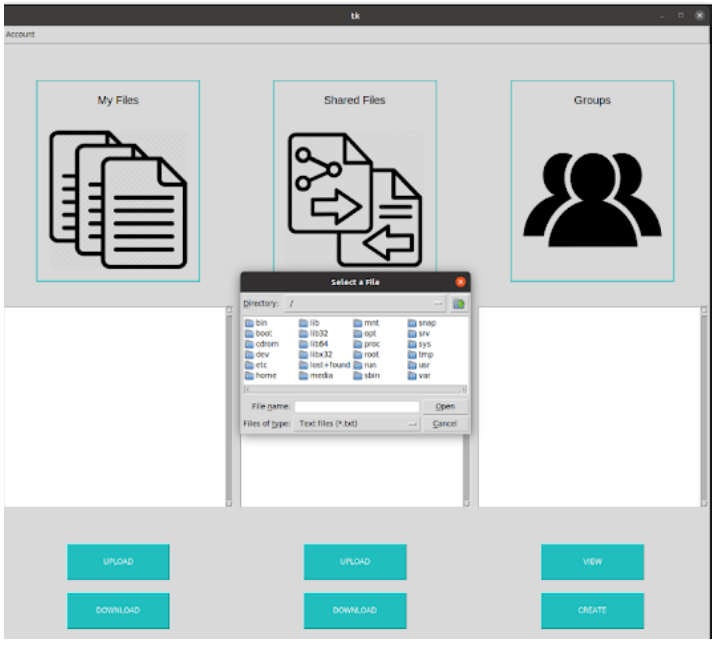
My Files



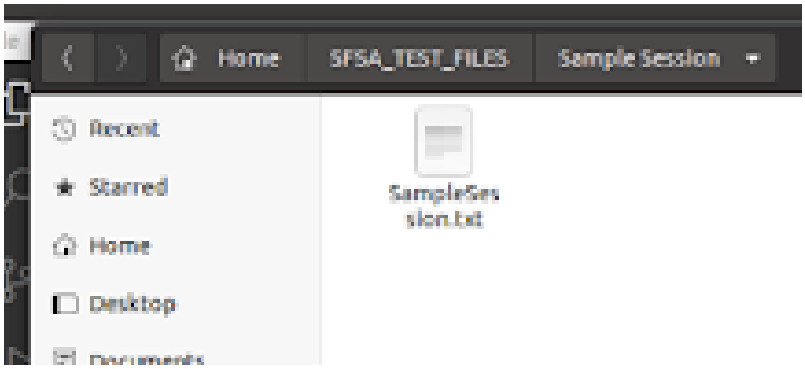
This section will refer to the above section, My Files, which can be found in the Main Screen.



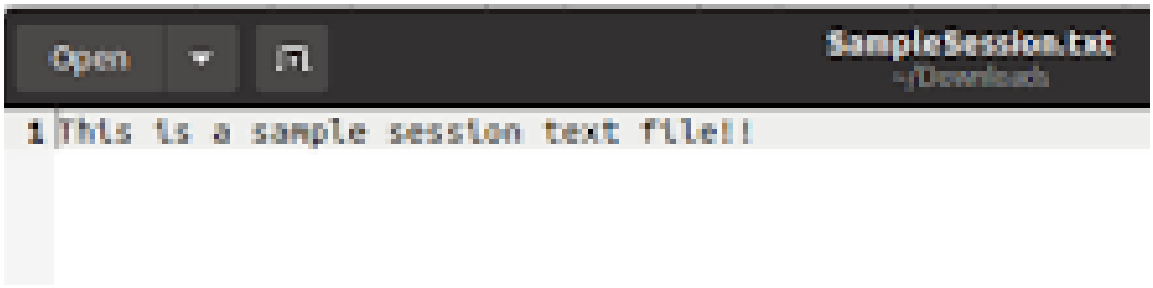
After pressing the ‘DOWNLOAD’ button in the My Files tab, the user is greeted with the above screen. No Files have been uploaded yet.



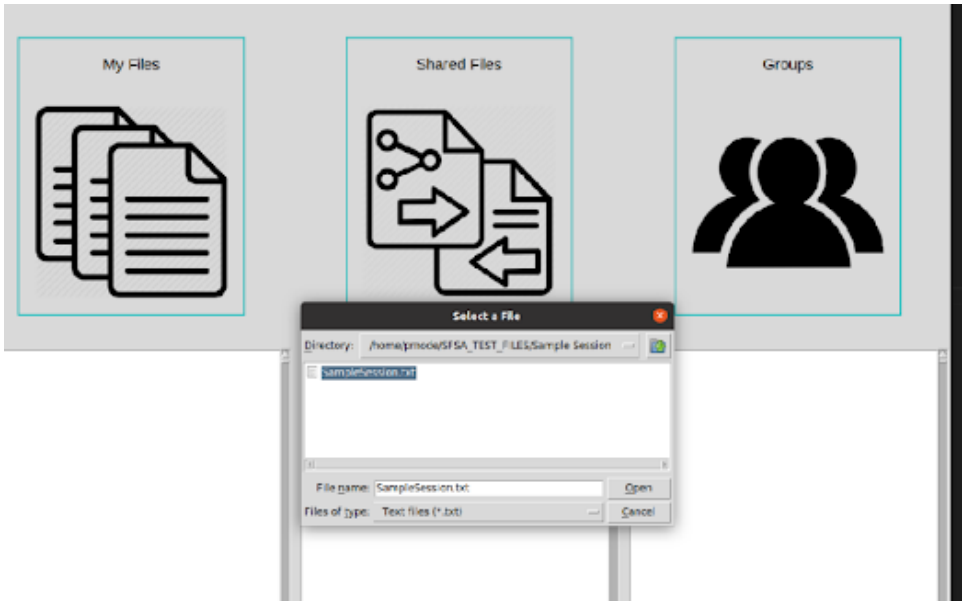
After pressing the ‘UPLOAD button in the My Files tab, the user is greeted with the above screen.



The above file is what we are going to upload to SFSA for this sample session.

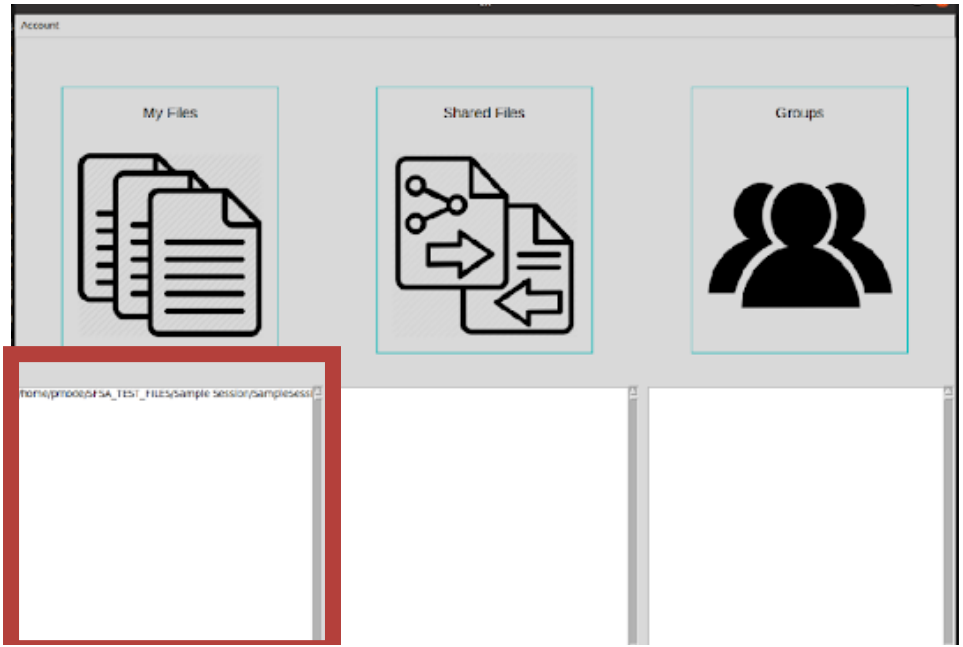


This is the text that is in the file SampleSession.txt



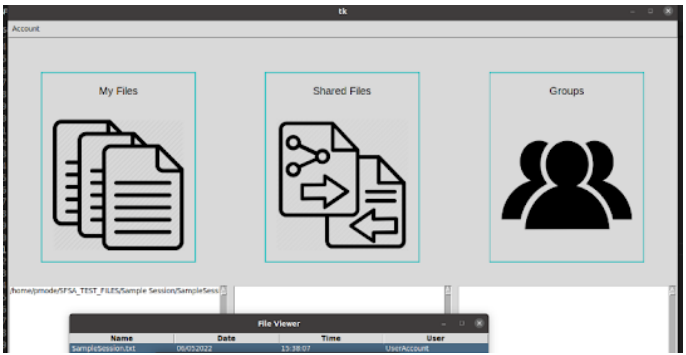
We navigate to the file shown previously to upload it.

After pressing open in the file explorer, the file is now encrypted and uploaded to SFSA.

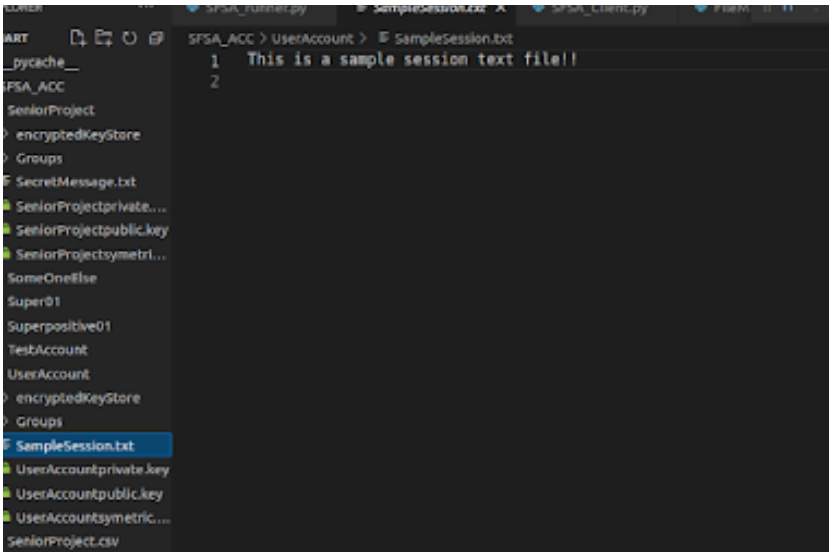


After pressing open in the file explorer, the window tab in My Files is updated to reflect what file we just uploaded.

After pressing the ‘DOWNLOAD’ button, we see the following screen in this session. We can see that our SampleSession.txt is uploaded and ready to download.

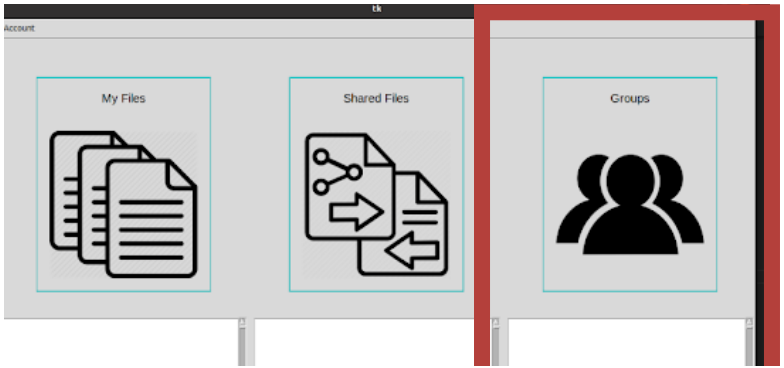


After double clicking on the file, we are greeted with the above screen.

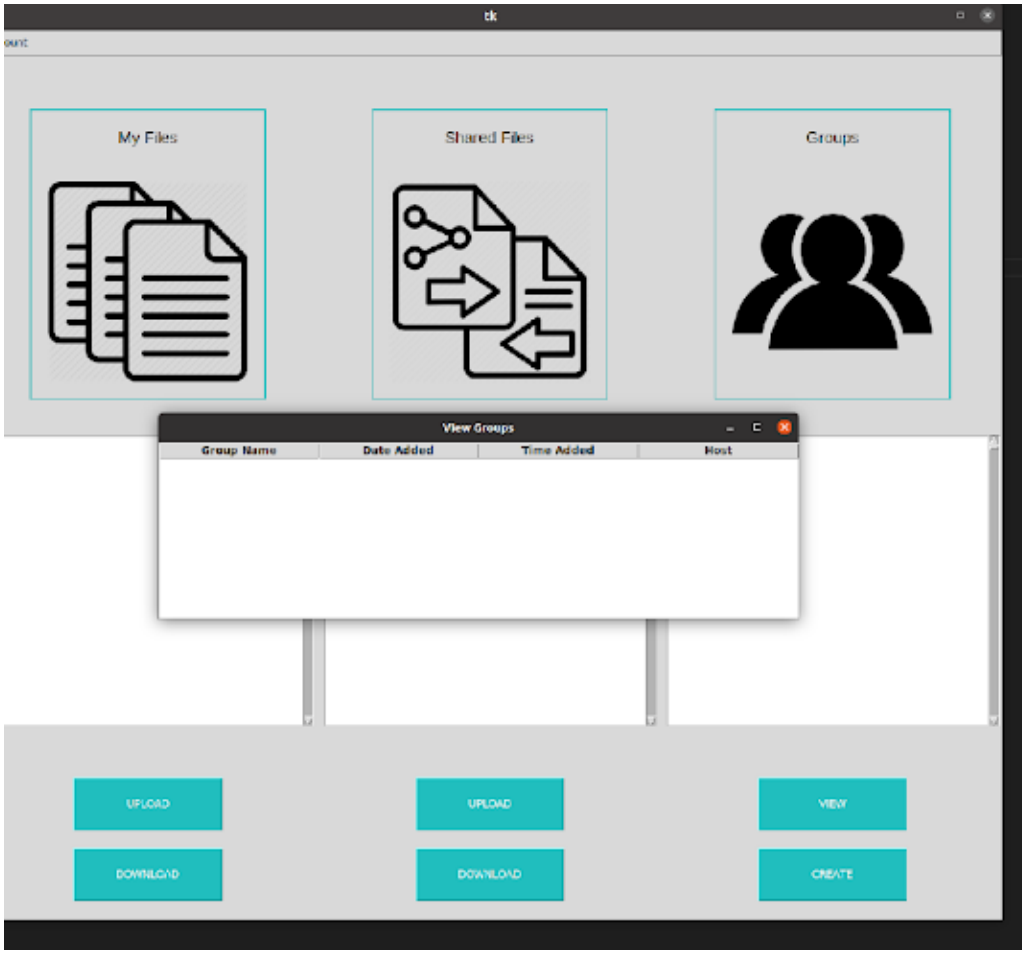


After clicking download we see that our file is downloaded in the working directory for the user and is decrypted.

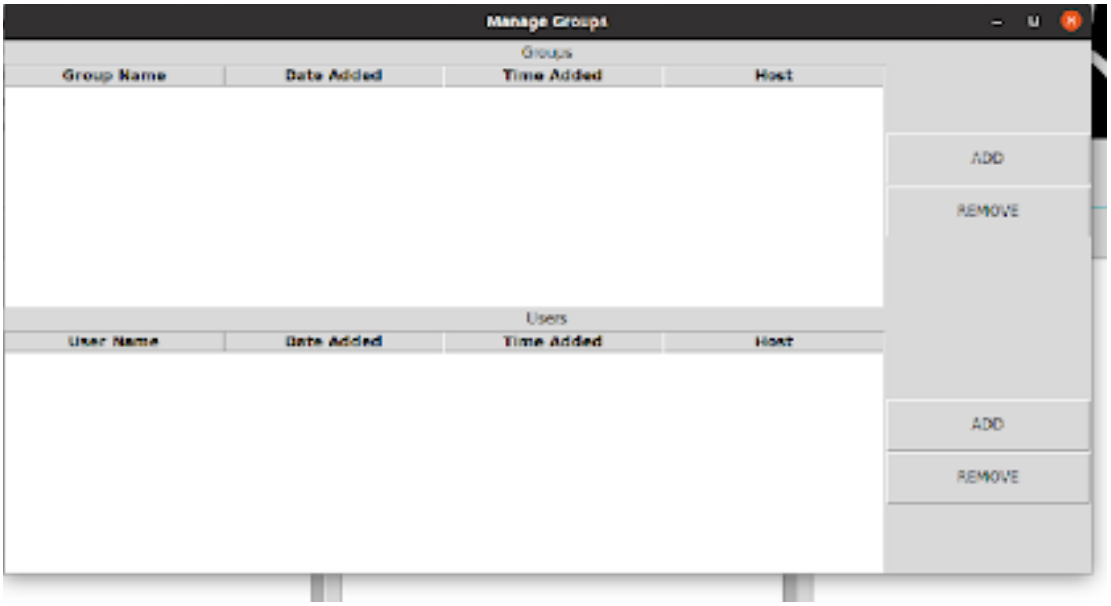
Groups



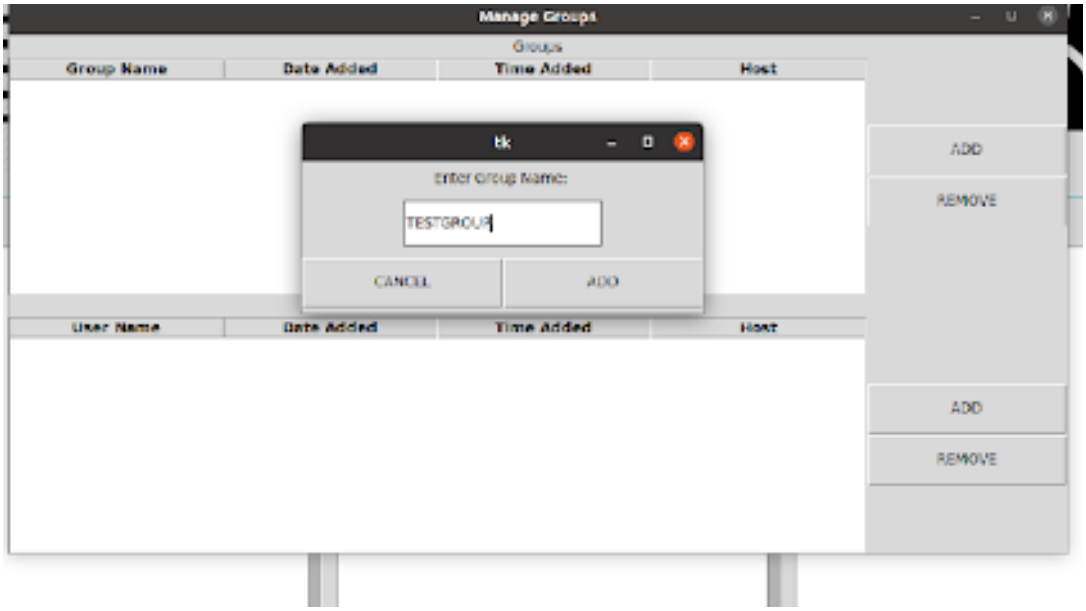
In this section we are focusing on the highlighted area on the main screen.



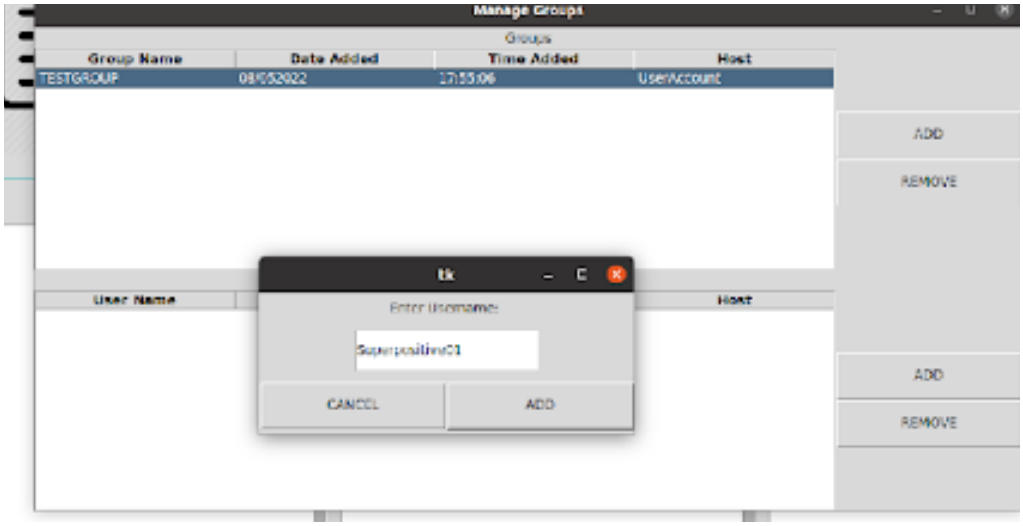
We can view our current groups by clicking the View Button. Currently in this session there are no new groups this user created.



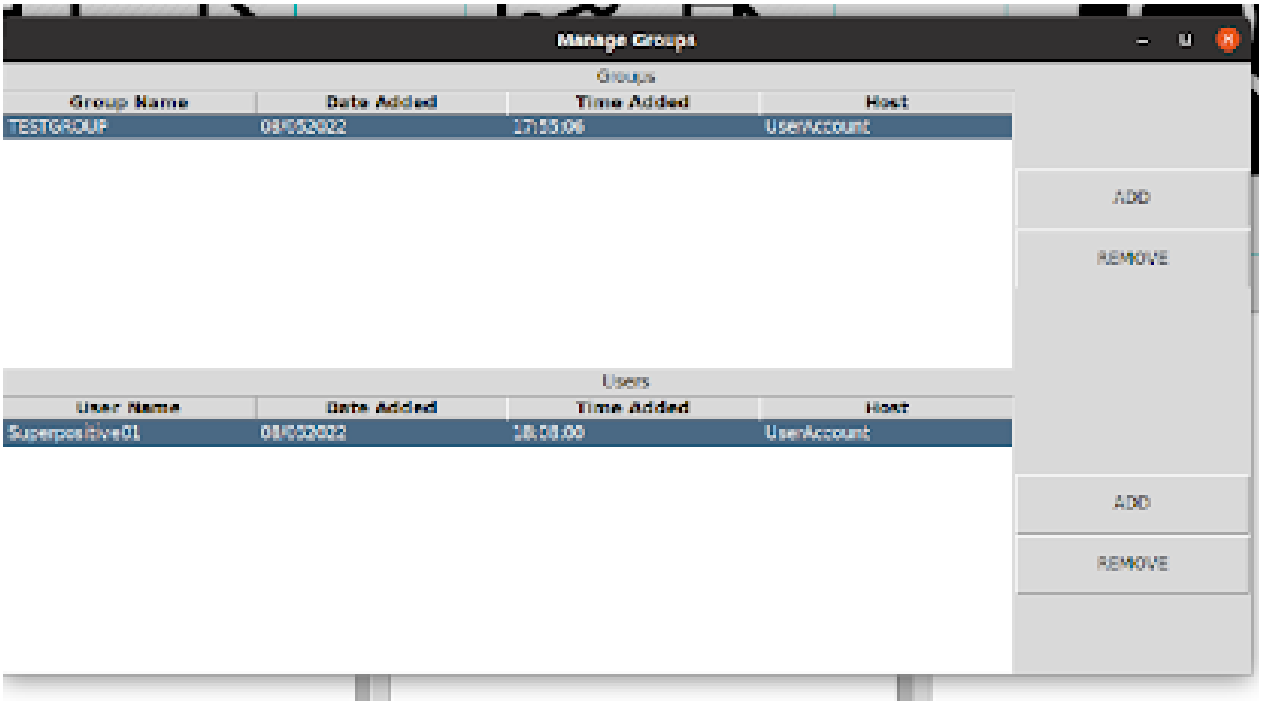
After clicking the CREATE button we see the above screen.



We click the top right ADD button and get the following popup. We are creating a group called TESTGROUP.



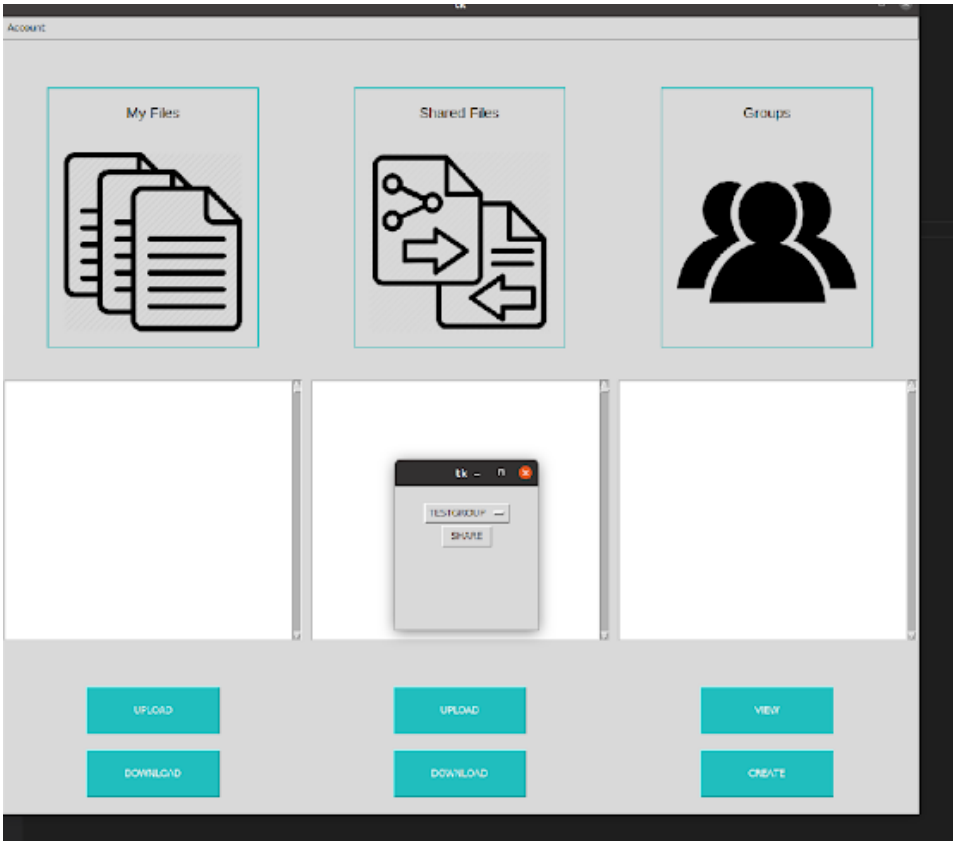
We click the top right ADD button and get the following popup. We are creating a group called TESTGROUP.



The user is added to the group. For all interactions for this screen look at the TEST PLAN.

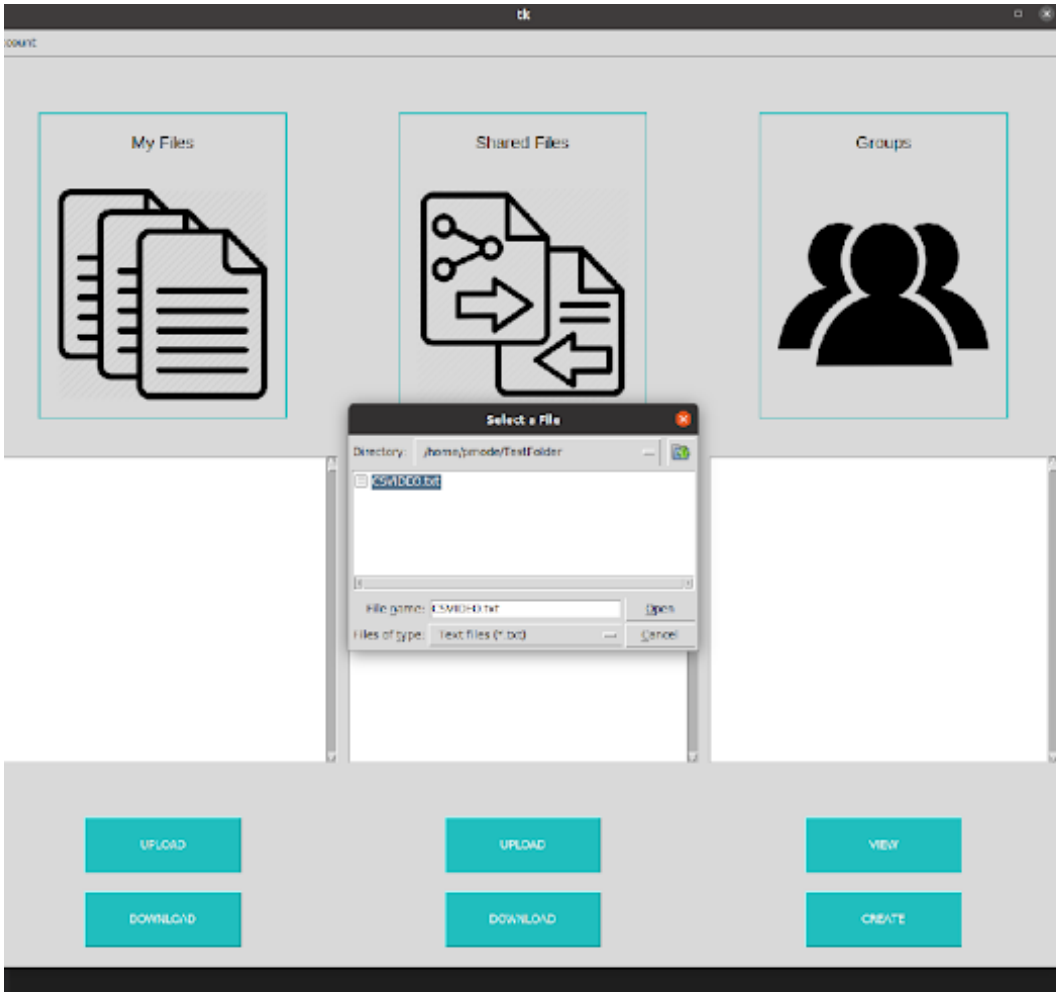
Shared Files

This section of the sample session will go over the last tab which is the share files tab.

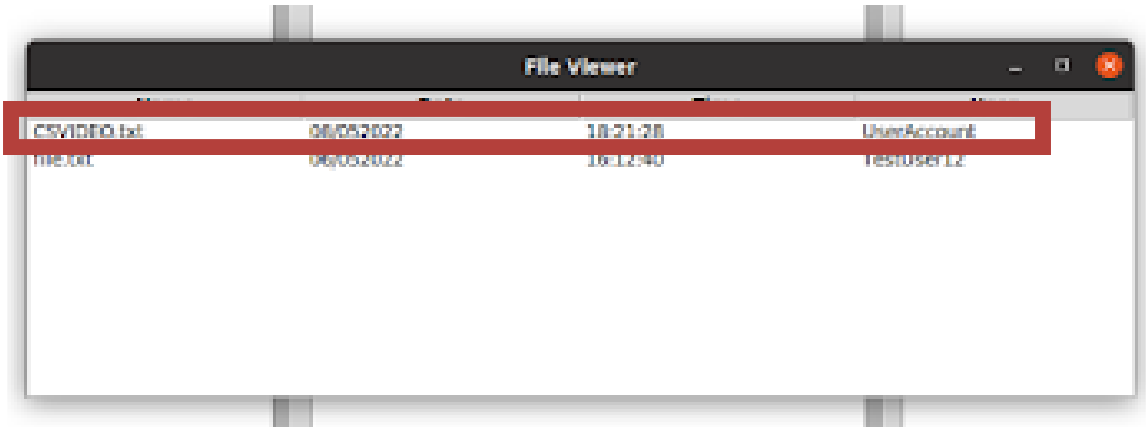


Clicking the UPLOAD button will make the above screen pop up. Here we can see a drop down menu of groups that this account made.

We select TESTGROUP FROM the drop down menu and click on the SHARE button.



After we click on the SHARE button we get a file explorer that works the same as the My Files Explorer. We then upload the current file and share it with the other user.



We login to the Account Superpositive01 and click DOWNLOAD. We see that the first entry is the shared file!

TROUBLESHOOTING

This section is dedicated to troubleshooting in the application and features common errors and solutions to those errors. Currently to run SFSA you need to have Ubuntu installed on two machines. One machine will run the server component while the other will run the client component. A more in-depth analysis of this page is written in the TEST PLAN for SFSA.

The following table shows errors that you might encounter using this product and suggests how to address each one.

Problem/Symptom	**Possible Cause**	**Solutions**
Failed to boot up. Missing IPFS daemon.	User did not start IPFS daemon 0.7.0 on their machine before starting IPFS.	Start IPFS daemon before running the server and client for SFSA.
Keys not showing up when logging in	Creation of a new account and logging in with a new account.	Login twice to your new account so the keys can be generated.
Can't decrypt upload file.	Unsupported document type.	SFSA currently only supports .txt plain text files. More file support will be added in the future.
Login Failed/ Upload Failed.	Packet loss from Client and Server sending messages.	Restart both client and server to send/receive messages again. Currently there is no support against packet loss.
Can't Create Account	Invalid credentials.	Make sure you meet the Account requirements for creating a account. These are listed in the test plan.

Client does not connect to server	Server not started or needs to be restarted.	Make sure to run the server before initialing a client session.

References

Binance Academy. “Byzantine Fault Tolerance Explained.” *Binance Academy*, Binance Academy, 24 Aug. 2021, https://academy.binance.com/en/articles/byzantine-fault-tolerance-explained.

Demiro. “Blockchain Public / Private Key Cryptography in a Nutshell.” *Medium*,

Coinmonks, 15 Nov. 2021,
https://medium.com/coinmonks/blockchain-public-private-key-cryptography-in-a-nutshell-b7776e475e7c.

“Ethereum Development Documentation.” *Ethereum.org*,
https://ethereum.org/en/developers/docs/.

Filecoin. “Build.” *Filecoin*, https://filecoin.io/build/#usp.

“How Decentralized Storage Works.” *How Decentralized Storage Works*, https://www.storj.io/how-it-works.

“Learn.” *Sia*, https://sia.tech/learn.

Sentdex. “Sockets Tutorial with Python 3.” *Python Programming Tutorials*,
https://pythonprogramming.net/sockets-tutorial-python-3/.

Tkinter 8.5 Reference: A Gui for Python, https://tkdocs.com/shipman/.

“Welcome to the Ipfs Docs.” *IPFS Docs*, https://docs.ipfs.io/.

Zhu, Zhiqin, et al. “Blockchain Based Consensus Checking in Decentralized Cloud Storage.” *Simulation Modelling Practice and Theory*, vol. 102,
https://doi.org/https://doi.org/10.1016/j.simpat.2019.101987.

Source Code File Listings

Accounts.py

```
# This file contains all the code to create a Account Filesystem
# Preston Robbins
# JAN 13 2022

from dataclasses import dataclass
from typing import List
import string


# data class for account creation
@dataclass
class Account:
        username: string
        password: string
        #privateKey: string
        #publicKey: string
        #keyRing : list
        #fileHashes: list
        #fileName: list

        #initalization function
        def __init__(self,name: str, password: str):
                self.name = name
                self.password = password

        # set the username
        def setUserName(self,name: str):
                self.username = name

        # get the user name
        def getUserName(self) -> string:
                return self.username

        #set the password
        def setPassword(self,pw : str):
                self.password = pw
```

```
        #get the password
        def getPassword(self) -> string:
                return self.password
```

AccountSystem.py

```
# AccountSystem.py
# Preston Robbins
# This file containsn functions that deal with account management

import re
from unittest import result


# Check if AccountName meets requirments
# Name no more than 25 characters
# Contains no special characters

def checkNameRequirments(accountName):
    #regular expression that checks for uppercase lowercase and numbers
    requirement = re.compile('^[A-Za-z0-9]+$')

    #result will return a match object if valid
    #if not then it will return None
    results = re.match(requirement,accountName)

    #check length requirment
    if(len(accountName) > 25):
        return False

    #check if no match is found
    elif(results == None):
        return False
```

```
    else:
        return True

# Check if Password meets requirments
# Password must be at least 8 characters
# Contains 1 special character
# Contaisn 1 Uppercase character
# No Spaces

def checkPasswordRequirement(password):

    #regular expression that checks for special character
    #checks for a number
    #checks for a uppercase character
    requirment =
re.compile("^(?=.*[\d])(?=.*[A-Z])(?=.*[a-z])(?=.*[@#$!])[\w\d@#$]{6,12}$"
)
    results = re.match(requirment,password)

    #check length requirment
    if(len(password) > 30):
        return False

    elif(results == None):
        return False

    else:
        return True

# Check if ConfirmBoxes are equal to each other

def confirmEqual(string1,string2):

    if(string1 == string2):
        return True

    else:
        return False

# get entry function
```

```
def getEntry(string1):
    return string1

# Confirm that userName is available

# Send request to server

# Get result of request from server
```

FileManager.py

```
# This program contains functions that will input
# data about a file that is shared between users of the program
# Each file will be named after the user and contains the following
# 1. FileName Shared
# 2. FileHash after encrpytion
# This is going to be in a csv file
# This will also create public/private keys for file sharing

from cmath import pi
from pathlib import Path
import csv
import shutil
from tabnanny import check
from tokenize import group
import rsa
from cryptography.fernet import Fernet
import os
#import ipfsApi
#import ipfsApi
import ipfshttpclient
import time
from datetime import date, datetime
import SFSA_Client
import pickle
```

```
import codecs
import pandas as pd

#api for ipfs
api = ipfshttpclient.connect('/ip4/10.0.2.15/tcp/5001')


#variable for current user filename and hash

u_name = ""
l_filename=""
l_hash=""

#path
path="SFSA_ACC/"

#add group header when we implement groups so we can use pandas dataframe
regular expressions to filter out who can access things
# The only thing the panel buttons will change is the flag
# also send keys to decrypt data across the wire

#header for csv
header = ['name','filename','hash','date','time','group']
#data to input in csv has to be in the following format
# user, filename, hash, date , time
#initalize to nothign when starting

data = [None] * 6

# set current filename
def setFilename(filename):
    global l_filename
    l_filename = filename

# get current filenmame
def getFileName():
    return l_filename

# set current hash
def setCurrentHash(hash):
    global l_hash
```

```
    l_hash = hash

# get current hash
def getHash():
    return l_hash

#After login store the current uer name in u_name
def setCurrentUser(name):
    global u_name
    u_name = name
#Get current user

def getCurrentUser():
    return u_name
#check for file in local directory that is u_name.txt
#do this on login
def checkForFile(name):
    filename = name.strip()
    filepath = f"SFSA_ACC/{filename}"
    filepath = Path(filepath)
    print(filepath)
    print(filepath.is_file())
    return filepath.is_file()


#create the file for login if it does not exsist
def createUserNameFile(name):
    filename = name +".csv"
    filepath = Path(path+filename)
    if checkForFile(filename) == False:
        with open(filepath, 'w', encoding='UTF8') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow(header)

            csvfile.close()

#place keys


def placeKeys(username,keyArr):
    privateKey = keyArr[0]
```

```
    publicKey = keyArr[1]
    symetricKey = keyArr[2]


    pKey = username + "private" + ".key"
    puKey  = username+ "public"  + ".key"
    symeKey = username + "symetric" + ".key"

    keyPath = f"SFSA_ACC/{username}/"




    #create key files
    with open(keyPath + f"{pKey}",'w') as f:
        f.write(privateKey)
        f.close()

    with open(keyPath + f"{puKey}",'w') as f:
        f.write(publicKey)
        f.close()
    with open(keyPath + f"{symeKey}",'w') as f:
        f.write(symetricKey)
        f.close()



#check if keys for user already exsist
def checkKeys(name):
    privateKey = name + "private" + ".key"
    publicKey  = name + "public"  + ".key"
    symetricKey = name + "symetric" + ".key"

    keyPath = f"SFSA_ACC/{name}/"

    privateKeyPath = Path(keyPath+privateKey)
    publicKeyPath = Path(keyPath+publicKey)
    symetricKeyPath = Path(keyPath+symetricKey)
```

```
    if (privateKeyPath.is_file() and publicKeyPath.is_file() and
symetricKeyPath.is_file()) == True:
        print("Keys Are GOOD!")
        return True

    else:
        print("Keys FALSE")
        return False


# create symetric key
# create public key
# create private key
def createKeys(name):



    #create account key directory


    if checkKeys(name) == False:

        keyPath = f"SFSA_ACC/{name}/"
        keypathos = os.path.join(path,name)

        os.mkdir(keypathos)

        encyptedKeyPath = f"SFSA_ACC/{name}/encryptedKeyStore/"


        os.makedirs(Path(encyptedKeyPath))

        #generate symetric key
        #key = Fernet.generate_key()
        #k = open(keyPath+f'{name}symetric.key','wb')
        #k.write(key)
        #k.close()

        #create the pub and private key
```

```
        #(pubkey,privkey) = rsa.newkeys(2048)
        # generate public and private keys
        #pukey = open(keyPath+f'{name}public.key','wb')
        #pukey.write(pubkey.save_pkcs1('PEM'))
        #pukey.close()

        #prkey = open(keyPath+f'{name}private.key','wb')
        #prkey.write(privkey.save_pkcs1('PEM'))
        #prkey.close()
        #print("CREATED KEYS")


# To share files encrpyt with a shared users public key
# To keep files to self encrpyt with current accounts public key

# send encrypted file and encrypted key so a user can decrypt the file

 #encrypt file

def encryptFile(name,file,group):
    #open symetric key
    keyPath = f"SFSA_ACC/{name}/"
    keystorePath = keyPath + "encryptedKeyStore/"

    skey = open(keyPath + f'{name}symetric.key','rb')
    key = skey.read()
    #create cipher
    cipher = Fernet(key)

    # open file for encrypting
    myfile = open(file, 'rb')
    myfiledata = myfile.read()
    #encrypt the data

    encrypted_data = cipher.encrypt(myfiledata)
    edata = open(file,'wb')
    edata.write(encrypted_data)
    edata.close()
    time.sleep(10)
```

```
    print(encrypted_data)

    #upload file to ipfs and save the hash of the file
    addTOIPFSNetwork(file)

    # open public key file
    pkey = open(keyPath + f"{name}public.key",'rb')
    pkdata = pkey.read()

    # load the file
    pubkey = rsa.PublicKey.load_pkcs1(pkdata)

    # encrypt the symmetric key file with the public key
    encrypted_key = rsa.encrypt(key,pubkey)

    # write encypted symetric key to a file
    #regular expression to get rid split file on all /
    ridOfPath = file.split('/')
    ridOfPath = ridOfPath[-1].strip()

    #set the filename
    setFilename(ridOfPath)

    ekey = open(keystorePath + f'{ridOfPath}encrypted.key','wb')
    ekey.write(encrypted_key)

    #send encrypted key to server
    ekey_data = encrypted_key

    #pickle message data


    server_message = [name,ekey_data,ridOfPath]

    server_message = pickle.dumps(server_message)

    SFSA_Client.setByteType("UPLOAD_SYM_KEY")

    SFSA_Client.addMessagetoQ(server_message)

    time.sleep(5)
```

```
    # add data to csv

    if group == 'self':
        addtoCSV(data,getCurrentUser(),group)

    elif group != 'self':

        #path to group selected
        pathToGroupFile =
f"SFSA_ACC/{getCurrentUser()}/Groups/{group}.csv"

            #open csv and send file to master
        with open(pathToGroupFile) as f:
                reader = csv.DictReader(f, delimiter=',')
                for row in reader:
                    name = row['user']
                    date = row['date']
                    u_time = row['time']
                    host = row['host']
                    addtoCSV(data,name,group)
                f.close()
    #print(encrypted_key)

# add file to ipfs network and save file hash to csv
def addTOIPFSNetwork(file):
    # add file to ipfs and return hash
    new_file = api.add(file)
    #get the hash of file to upload to blockchain
    #0 - name 1 - hash 2- size what is reutrned when adding to ipfs


    #print(new_file)
    fileHash = new_file['Hash']

    #file_hash = new_file[0]
    #file_hash_value = file_hash.get('Hash')

    setCurrentHash(fileHash)
    print(fileHash)
```

```
#function to get hash from csv file
#also downloads file from ipfs and changes the name to original txt
#also calls decrpytion
def getHashFromCSV(name,filename,date,time,group):

    csvPath = f"SFSA_ACC/{name}.csv"

    #convert csv into pandas dataframe
    df = pd.read_csv(csvPath)


    #get rows hash that is equal to name,filename,date,time,group
    hashValueDF = df.loc[ ( (df['name'] == name) & (df['filename'] ==
filename) & (df['date'] == date) &  (df['time'] == time) & (df['group'] ==
group) )]

    #set current row in data frame to hash
    hash = hashValueDF['hash'].item()
    hash = hash.strip()

    #return hash

    #call get from IPFS based on hash we have
    getFROMIPFSNetwork(hash,name)

    #call file rename also calls decrpytion
    renameFilePulledFromIPFS(hash,name)



# get file from IPFS newtork
def getFROMIPFSNetwork(hash,name):
    #use the api to get file from hash
    api.get(hash)
    #move to SFSA_ACC
    shutil.move(hash,f"SFSA_ACC/{name}/{hash}")
```

```
def renameFilePulledFromIPFS(hash,name):
    #working directory
    directory = f"SFSA_ACC/"
    #rename file to what is in the csv
    df = pd.read_csv(directory+f"{name}.csv")

    fileNameDF = df.loc[(df['hash'] == hash)]

    filename = fileNameDF['filename'].item()
    #rename the file
    os.rename(directory +f"{name}/{hash}",directory +f"{name}/{filename}")

    #decrpyt the file
    decryptFile(name,filename)

    #get row with hash
# decrypt file
def decryptFile(name,file):
     #open symetric key
    keyPath = f"SFSA_ACC/{name}/"

    #load the priave key to decrypt the public key
    prkey = open(keyPath + f'{name}private.key', 'rb')
    pkey = prkey.read()
    private_key = rsa.PrivateKey.load_pkcs1(pkey)

    #wopen encrypted key
    e = open(keyPath+"encryptedKeyStore/" + f'{file}encrypted.key', 'rb')
    ekey = e.read()
    dpubkey = rsa.decrypt(ekey,private_key)

    #create cipher
    cipher = Fernet(dpubkey)

    #open encrypted data
    encrypted_data = open(keyPath+f"{file}",'rb')

    edata = encrypted_data.read()
    decrypted_data = cipher.decrypt(edata)
```

```
    #print(decrypted_data.decode('unicode_escape'))
    #open file name and write to it
    file = open(keyPath+f"{file}",'w')
    file.write(decrypted_data.decode())
    file.close()


    #open encrypted key
    #tempory place where keys are

#add if statement if is apart of group or not
#default group is self

#get Symetric Key

def sendMsgTogetSymetricKey(username,filename):

    SFSA_Client.setByteType("REQUEST_SYMKEY")

    info = [username,filename]

    info = pickle.dumps(info)

    SFSA_Client.addMessagetoQ(info)


# function to add to csv
def addtoCSV(data,name,group):
    #set indexs of data so we can append our row to csv and create a
dataframe to check for uploads later
    today = datetime.now()
    dt_string = today.strftime("%d/%m%Y %H:%M:%S")
    dt_arr = dt_string.split(" ")
    c_date = dt_arr[0]
    c_time = dt_arr[1]


    #use pickle for data to send over the wire
```

```
    if name == getCurrentUser():
        data[0] = name
        data[1] = getFileName()
        data[2] = getHash()
        data[3] = c_date
        data[4]  = c_time
        data[5]  = group
        message = pickle.dumps(data)

        print("TYPE of message CSV",type(message) )

        SFSA_Client.setByteType("APPEND_USER_DOWNLOAD")


        #send the message add file user ADDFLU with data attachted and
send data array to server
        SFSA_Client.addMessagetoQ(message)

    elif name != getCurrentUser():
        #print("TYPE of message CSV",type(message) )

        #append original user to data
        nameToSplit = name + " " + getCurrentUser()
        data[0] = nameToSplit
        data[1] = getFileName()
        data[2] = getHash()
        data[3] = c_date
        data[4]  = c_time
        data[5]  = group

        message = pickle.dumps(data)

        SFSA_Client.setByteType("APPEND_USER_DOWNLOAD")

        #send the message add file user ADDFLU with data attachted and
send data array to server
        SFSA_Client.addMessagetoQ(message)

    # send data over the wire to server to append to master user file

    #write to csv file
```

```
    #filename = name +".csv"
    #filepath = Path(path+filename)

    #with open(filepath, 'a') as csvfile:
        #create writer object
    #    csvwriter = csv.writer(csvfile)

        #write row
    #    csvwriter.writerow(data)

    #    csvfile.close()


#function to make current user csv a pandas dataframe for querying

#read from csv to update gui
```

GroupManager.py

```
# This file is the python file that is responsible for managing groups and
interacts with the GUI
import os
from pathlib import Path
import csv
from datetime import date, datetime
from tokenize import group
import pandas as pd

#create group folder for user

#header for group file
#header for csv
header = ['name','date','time','host']
#header for csv users to add to group
userheader = ['user','date','time','host']


# data for group csv
```

```
data = [None] * 4

#also creates mastergroupfile textfile to keep track of all groups
def createGroupFolder(username):
    #file path to folder
    path = Path(f"SFSA_ACC/{username}/Groups/")

    if os.path.exists(path) == False:
        os.mkdir(path)
    else:
        print("GOOD PATH FOR USER")

    #check if mastergroup file is created
    filepath = Path(f"SFSA_ACC/{username}/Groups/masterfile.csv")
    print("FILEPATH GROUPS IS", filepath)
    if filepath.is_file() == False:
        #create the file
        with open(filepath,'w') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow(header)
            csvfile.close()


#add to group masterfile
def addToGroupMasterFile(username,groupname):
    path = f"SFSA_ACC/{username}/Groups/masterfile.csv"

    #for date and time
    today = datetime.now()
    dt_string = today.strftime("%d/%m%Y %H:%M:%S")
    dt_arr = dt_string.split(" ")
    c_date = dt_arr[0]
    c_time = dt_arr[1]
    #the fileds for the file are name, date ,time ,host


    data[0] = groupname
    data[1] = c_date
    data[2] = c_time
    data[3] = username
```

```
    #append to csv
    with open(path, 'a') as csvfile:
        #create writer object
        csvwriter = csv.writer(csvfile)

        #write row
        csvwriter.writerow(data)

        csvfile.close()

def removeFromGroupMasterFile(username,name,date,time,host):
    #path to groupfile
    path = f"SFSA_ACC/{host}/Groups/masterfile.csv"

    #convert csv to dataframe
    df = pd.read_csv(path)

    #locate row with groupname
    #groupValueDF = df.loc[  ( (df['name'] != name) & (df['date'] != date)
& (df['time'] != time) &  (df['host'] != host) )]
    groupValueDF = df.loc[  ( (df['name'] != name))]

    print("GROUP DATA FRAME IS ", groupValueDF)

    #convert df to modified csv
    groupValueDF.to_csv(path,index=False)
    #remove file from user directory
    removeGroup(host,name)

#function that adds groups in the format groups.txt

def addGroup(username,groupName):

    path = f"SFSA_ACC/{username}/Groups/"
    # go to file path form username

    group_text = groupName.strip()+".csv"

    group_file = Path(path+group_text)
```

```
    #create group name if it is not a duplicate

    if group_file.is_file() == False:
        #create group text file
        with open(group_file,'w') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow(userheader)
            csvfile.close()

    #call to append group to group master file to make it visable on gui
    addToGroupMasterFile(username,groupName)

def removeGroup(username,groupName):
    #Path to CSV
    path = f"SFSA_ACC/{username}/Groups/"

    #append .csv to groupName
    filename = Path(f"{path}+{groupName}.csv")

    #delete the file if it exsits
    if filename.is_file() == True:
        os.remove(filename)

def addUserToGroupFile(username,group,user):
    #path to group csv
    path = f"SFSA_ACC/{username}/Groups/{group}.csv"

    #for date and time
    today = datetime.now()
    dt_string = today.strftime("%d/%m%Y %H:%M:%S")
    dt_arr = dt_string.split(" ")
    c_date = dt_arr[0]
    c_time = dt_arr[1]
    #the fileds for the file are name, date ,time ,host


    data[0] = user
    data[1] = c_date
    data[2] = c_time
    data[3] = username
```

```
    #append row to

    #append to csv
    with open(path, 'a') as csvfile:
        #create writer object
        csvwriter = csv.writer(csvfile)

        #write row
        csvwriter.writerow(data)

        csvfile.close()

def removeUserFromGroup(username,group,user):
    #path to groupfile
    path = f"SFSA_ACC/{username}/Groups/{group}.csv"

    #convert csv to dataframe
    df = pd.read_csv(path)

    #locate row with groupname
    groupValueDF = df.loc[  ( (df['user'] != user))]

   #print("GROUP DATA FRAME IS ", groupValueDF)

    #convert df to modified csv
    groupValueDF.to_csv(path,index=False)

#def appendToMasterFile(name):
    #get path to masterfile
#    path = f"SFSA_ACC/{name}/Grops/masterfile.csv"
```

SFSA.py

```
# This file contains all layouts for SFSA GUI
# Preston Robbins
# JAN 13 2022

# Multi-frame tkinter application v2.3
from cProfile import label
```

```
from email import message
from email.headerregistry import Group
from fileinput import filename
#from msilib.schema import File
from pathlib import Path
import tkinter as tk
from tkinter.constants import SEL_FIRST, W
from tkinter.tix import Tree
from tokenize import group
from typing import Container
from PIL import Image, ImageTk
from tkinter import Menu, ttk
from tkinter import filedialog
from tkinter import messagebox
import threading

from numpy import gradient
from requests import options
from AccountSystem import *
import Accounts
import SFSA_Client
import time
import FileManager
import GroupManager
import csv

class SFSA(tk.Tk):


    #initialization function for frames
    def __init__(self, *args, **kwargs):

        tk.Tk.__init__(self,*args,**kwargs)

        # set teh container to tk.Frame
        container = tk.Frame(self)

        #pack
        container.pack(side="top",fill="both",expand=True)
```

```
        container.grid_rowconfigure(0, weight=1)
        container.grid_columnconfigure(0,weight=1)


        screens = [WelcomeScreen,LoginScreen,CreateAccountScreen,
MainMenu]

        #dictonary of frames
        self.frames = {}

        for F,geometry in zip
((WelcomeScreen,LoginScreen,CreateAccountScreen, MainMenu),
('1850x554','1280x1080','1280x1080','1280x1080')):

            # set frame to start page upon initalization
            frame = F(container,self)

            # set current frame to StartPage
            self.frames[F] = (frame,geometry)


            frame.grid(row=0, column=0, sticky="nsew")

        SFSA.showMenuBar(self,container)

        # show the frame
        self.show_frame(WelcomeScreen)
        self.bind('<Button-1>', SFSA.display_coordinates)

    #print x y coordiantes upon button press
    def display_coordinates(event):
            print(f'x={event.x} y={event.y}')

    # function to show frame
    def show_frame(self,cont):
        frame,geometry = self.frames[cont]
        self.geometry(geometry)
        frame.tkraise()

    # function to create menu bar that shows options
    def showMenuBar(self,container):
```

```
        # Create a menubar with options such as 'Account' -> 'LOGOUT'

        menuBar = tk.Menu(container)
        fileMenu =  tk.Menu(menuBar, tearoff=0)

         #Add logout button under Account in menu

fileMenu.add_command(label="Logout",command=SFSA.confirmLogoutPopup)
        menuBar.add_cascade(label="Account", menu=fileMenu)

        #configure self to have these options
        tk.Tk.config(self,menu=menuBar)

    def confirmLogoutPopup():

        messagebox.askyesno('Yes|No', 'Do you want to logout?')


class WelcomeScreen(tk.Frame):
        def __init__(self,parent,controller):

            tk.Frame.__init__(self,parent)

            # set the window size for the screen
            canvas = tk.Canvas(self, width=1850, height=554)

            # use the 3 column grid
            canvas.grid(columnspan=3)


            #open image using PIL library
            logo = Image.open('loginscreen.png')

            #convert image to a usable tkinter library
            logo = ImageTk.PhotoImage(logo)

            # Set image in canvas
            canvas.create_image(0,0, image=logo, anchor="nw")

            canvas.create_image = logo
```

```
            # Login Button
            login_btn = tk.Button(self, text="LOGIN", anchor = 'center' ,
font="Raleway", bg ="#20bebe", fg="white", height=2, width=17,
command=lambda:controller.show_frame(LoginScreen))

            login_btn_window = canvas.create_window(947,403, anchor="sw",
window=login_btn)

            create_acc_btn = tk.Button(self, text="CREATE ACCOUNT", anchor
= 'center' , font="Raleway", bg ="#20bebe", fg="white", height=2,
width=17, command=lambda:controller.show_frame(CreateAccountScreen))

            create_acc_window = canvas.create_window(947,508, anchor="sw",
window=create_acc_btn)

class LoginScreen(tk.Frame):

    #variable to when to switch to main menu frame if valid login
    switchFrame = False

    #send login info to server
    def sendLoginInfo(username,password):
            message = username + " " + password
            FileManager.setCurrentUser(username)
            SFSA_Client.addMessagetoQ("LOGINX " + message)

   #function to switch frames to main app
    def showMainMenu(self, controller, username, password):

        username = username.strip()
        LoginScreen.sendLoginInfo(username,password)

        time.sleep(5)
        if LoginScreen.switchFrame == True:
            FileManager.setCurrentUser(username)
            controller.show_frame(MainMenu)
            LoginScreen.switchFrame = False
```

```
            # check for local file storage for this account

            FileManager.createUserNameFile(username)
            FileManager.createKeys(username)
            GroupManager.createGroupFolder(username)



    def __init__(self,parent,controller):

            tk.Frame.__init__(self,parent)

            #function that sends in all the infomration in the fields as a
message


            # Create a 3x3 grid to place our packed layout in

            self.grid_columnconfigure(0, weight=1)
            self.grid_columnconfigure(1, weight=1)
            self.grid_columnconfigure(2, weight=1)

            self.grid_rowconfigure(0,weight=1)
            self.grid_rowconfigure(1,weight=1)
            self.grid_rowconfigure(2,weight=1)
            self.grid_rowconfigure(3,weight=1)

            #master frame that we place in the grid we created for this
frame
            frame = tk.Frame(self)
            frame.grid(column=1,row=2)

            #frames to have icon and entry in the same area

            #contains userIcon and entry
            userEntryFrame= tk.Frame(frame)
            #contrains passwordIco and Entry
            passwordEntryFrame=tk.Frame(frame)

            #contains back and login buttons
            buttonsFrame = tk.Frame(frame)
```

```
            # Get login icons for screen
            userIcon = Image.open("user_Icon.png")
            passwordIcon= Image.open("lock_Icon.png")

            #SFSA Logo
            sfsa_logo = Image.open("SFSALOGO2.png")

            #convert icons to tkinter widget
            userI_tk = ImageTk.PhotoImage(userIcon)
            passI_tk = ImageTk.PhotoImage(passwordIcon)

            sfsa_logo = ImageTk.PhotoImage(sfsa_logo)

            #create labels for image
            userIconLabel = tk.Label(userEntryFrame,image=userI_tk)
            passIconLabel = tk.Label(passwordEntryFrame,image=passI_tk)

            sfsa_logoLabel = tk.Label(frame,image=sfsa_logo)

            userIconLabel.image = userI_tk
            passIconLabel.image = passI_tk
            sfsa_logoLabel.image = sfsa_logo

            # Username label
            usernameLabel = tk.Label(frame, text="Username",
font=("Arial",15))



            # Username entry
            usernameEntry = tk.Entry(userEntryFrame)


            # Password label
            passwordLabel = tk.Label(frame,
text="Password",font=("Arial",15))


            # Password entry
            passwordEntry = tk.Entry(passwordEntryFrame)
```

```
            #Login Button
            login_btn = tk.Button(buttonsFrame, text="LOGIN",
bg="#20bebe",  fg="white", anchor='center', height=2, width=20,
command=lambda:LoginScreen.showMainMenu(self,controller,usernameEntry.get(
),passwordEntry.get()))
            #login_btn = tk.Button(buttonsFrame, text="LOGIN",
bg="#20bebe",  fg="white", anchor='center', height=2, width=20,
command=lambda:controller.show_frame(MainMenu))

            back_btn = tk.Button(buttonsFrame, text="BACK", bg="#20bebe",
 fg="white", anchor='center', height=2, width=20,
command=lambda:controller.show_frame(WelcomeScreen))




            # Pack Logo
            sfsa_logoLabel.pack()

            #use pack to put in frame within master grid

            #pack label in frame
            usernameLabel.pack(ipadx=50, ipady=50)
            #pack Entry frame with userIconLabel and UserEntry
            userEntryFrame.pack()

            #pack userLabel and usernameEntry in Frame
            userIconLabel.pack(side=tk.LEFT)
            usernameEntry.pack(side=tk.LEFT,ipadx=10, ipady=10)

            #password label in sub frame
            passwordLabel.pack(side=tk.TOP,ipadx=50, ipady=50)

            #pack entry fraem with passwordicon and password entry
            passwordEntryFrame.pack()

            #pack widgets in passwordframe
            passIconLabel.pack(side=tk.LEFT, padx=10)
            passwordEntry.pack(side=tk.LEFT,ipadx=10, ipady=10)

            #pack budget frame in subframe
```

```
            buttonsFrame.pack()

            back_btn.pack(side=tk.LEFT, ipady=10, pady=20,padx=2)
            login_btn.pack(side=tk.LEFT, ipady=10, pady=20,padx=2)




class CreateAccountScreen(tk.Frame):


    #netsed function that checks requirements for account creation
    # takes in username confirmusername password confirmpassword
    def checkRequirments(un,c_un,pw,c_pw):

        if(checkNameRequirments(un) == False):

            messagebox.showerror(message="Username Requirements not
meet!")

        elif(checkPasswordRequirement(pw) == False):

            messagebox.showerror(message="Password Requirements not
meet!")

        elif(confirmEqual(un,c_un) == False):

            messagebox.showerror(message="Usernames do not match!")

        elif(confirmEqual(pw,c_pw) == False):

            messagebox.showerror(message="Passwords do not match!")

        else:
                #Create Account with UserName and Password
                newAcc = Accounts.Account
                newAcc.setUserName(newAcc,un)
                newAcc.setPassword(newAcc,pw)
                #print(newAcc.getUserName(newAcc))

                #Appened it to Client Roster
```

```
                username = newAcc.getUserName(newAcc)
                password = newAcc.getPassword(newAcc)
                message = username + " " + password
                SFSA_Client.addMessagetoQ("SIGNUP " + message)


    def __init__(self,parent,controller):

        tk.Frame.__init__(self,parent)

        #set the weights of the columns and rows
        self.grid_columnconfigure(0, weight=1)
        self.grid_columnconfigure(1, weight=3)
        self.grid_columnconfigure(2, weight=1)

        self.grid_rowconfigure(0,weight=1)
        self.grid_rowconfigure(1,weight=1)
        self.grid_rowconfigure(2,weight=1)
        self.grid_rowconfigure(3,weight=1)

        frame = tk.Frame(self)
        frame.grid(column=1, row=2)



         #contains userIcon and entry
        userEntryFrame= tk.Frame(frame)

         #contains userIcon and Confirm entry
        userEntryConfirmFrame= tk.Frame(frame)

        #contrains passwordIco and Entry
        passwordEntryFrame=tk.Frame(frame)

        #contarins passwordIco and Password confirm entry
        passwordConfirmFrame= tk.Frame(frame)

        #contains back and create account buttons
        buttonsFrame = tk.Frame(frame)
```

```
        # Get login icons for screen
        userIcon = Image.open("user_Icon.png")
        passwordIcon= Image.open("lock_Icon.png")

        keyImage = Image.open("key.png")
        #SFSA Logo
        sfsa_logo = Image.open("SFSALOGO.png")

        sfsa_logo = ImageTk.PhotoImage(sfsa_logo)

        #convert icons to tkinter widget
        userI_tk = ImageTk.PhotoImage(userIcon)
        passI_tk = ImageTk.PhotoImage(passwordIcon)
        keyI_tk = ImageTk.PhotoImage(keyImage)


        #create labels for image
        userIconLabel = tk.Label(userEntryFrame,image=userI_tk)
        passIconLabel = tk.Label(passwordEntryFrame,image=passI_tk)
        sfsa_logoLabel = tk.Label(frame,image=sfsa_logo)

        sfsa_logoLabel.image = sfsa_logo

        userIconLabel.image = userI_tk
        passIconLabel.image = passI_tk


        keyImageUserName = tk.Label(userEntryConfirmFrame,image=keyI_tk)
        keyImageUserName.image = keyI_tk

        keyImagePassword = tk.Label(userEntryConfirmFrame,image=keyI_tk)
        keyImagePassword.image = keyI_tk

        keyImagePasswordConfirm =
tk.Label(passwordConfirmFrame,image=keyI_tk)
        keyImagePasswordConfirm.image = keyI_tk

        #Username
        userNameLabel= tk.Label(frame, text="Username",font=("Arial",15))

        #Username entry
```

```
        userNameEntry= tk.Entry(userEntryFrame)

        #Confirm UserName
        confirmUserNameLabel = tk.Label(frame, text="Confirm
Username",font=("Arial",15))

        #Confirm UserName Entry

        confirmUserNameEntry = tk.Entry(userEntryConfirmFrame)

        #Password
        passwordLabel= tk.Label(frame,text="Password", font=("Arial",15))

        #Password entry
        passwordEntry= tk.Entry(passwordEntryFrame)

        #Confirm Password
        confrmPasswordLabel= tk.Label(frame,text="Confirm
Password",font=("Arial",15))

        #Confirm Password entry
        confirmPasswordEntry= tk.Entry(passwordConfirmFrame)

        #Create Account Button
        create_acc_btn = tk.Button(buttonsFrame, text="CREATE ACCOUNT",
bg="#20bebe",  fg="white", anchor='center', height=2, width=20,

command=lambda:CreateAccountScreen.checkRequirments(userNameEntry.get(),

confirmUserNameEntry.get(),

passwordEntry.get(),

confirmPasswordEntry.get()))

        #Back Button
        back_btn = tk.Button(buttonsFrame, text="BACK",  bg="#20bebe",
fg="white", anchor='center', height=2, width=20,
command=lambda:controller.show_frame(WelcomeScreen))
```

```
#Logo
sfsa_logoLabel.pack()

#pack contents into frame
userNameLabel.pack(ipadx=50, ipady=50)

#userEntryFrame
userEntryFrame.pack()

userIconLabel.pack(side=tk.LEFT)
userNameEntry.pack(side=tk.LEFT,ipadx=10, ipady=10)


#confirm UserName
confirmUserNameLabel.pack(ipadx=50, ipady=50)
userEntryConfirmFrame.pack()

keyImageUserName.pack(side=tk.LEFT)
confirmUserNameEntry.pack(side=tk.LEFT,ipadx=10, ipady=10)


#passwordFrame
passwordLabel.pack(ipadx=50, ipady=50)
passwordEntryFrame.pack()

passIconLabel.pack(side=tk.LEFT)
passwordEntry.pack(side=tk.LEFT,ipadx=10, ipady=10)

#confirm password frame
confrmPasswordLabel.pack(ipadx=50, ipady=50)

passwordConfirmFrame.pack()
keyImagePasswordConfirm.pack(side=tk.LEFT)
confirmPasswordEntry.pack(side=tk.LEFT,ipadx=10, ipady=10)

#buttons frame
buttonsFrame.pack()
back_btn.pack(side=tk.LEFT,ipady=10, pady=20,padx=2)
```

```
        create_acc_btn.pack(side=tk.LEFT,ipady=10, pady=20,padx=2)


class MainMenu(tk.Frame):


# ** MY FILES POPUP SCREENS
**************************************************************************
*******


    def downloadFileSelf(values):

        currentUser = FileManager.getCurrentUser()

        #get values from array
        filename = values[0].strip()
        date = values[1].strip()
        u_time = values[2].strip()
        username = values[3].strip()
        group = 'self'

        #get symetric key from server
        FileManager.sendMsgTogetSymetricKey(username,filename)

        time.sleep(10)

        FileManager.getHashFromCSV(currentUser,filename,date,u_time,group)

    def downloadFilesShared(values):
        currentUser = FileManager.getCurrentUser()

        #get values from array
        filename = values[0].strip()
        date = values[1].strip()
        u_time = values[2].strip()
        username = values[3].strip()
        group = 'other'

        #get symetric key from server
        FileManager.sendMsgTogetSymetricKey(username,filename)
```

```
        time.sleep(10)

        FileManager.getHashFromCSV(currentUser,filename,date,u_time,group)

    #creates a popup window for confirming file downlaod
    def confirmDownloadWindow(values):
        #create tkinter top level view of window
        confirmDownloadWindow = tk.Toplevel()

        #create label that shows file info
        labelText = f"{values[0]} {values[1]} {values[2]} {values[3]}"

        #label to confirm selected file
        fileInfoLabel = tk.Label(confirmDownloadWindow, text=labelText,
font=("Arial",15))

        #button for downloading files
        button = tk.Button(confirmDownloadWindow,text="DOWNLOAD" ,
anchor='center', height=2, width=20,
command=lambda:MainMenu.downloadFileSelf(values))

        fileInfoLabel.pack()

        button.pack()

    #creates a popup window for confirming file downlaod
    def confirmDownloadWindowShared(values):
        #create tkinter top level view of window
        confirmDownloadWindow = tk.Toplevel()

        #create label that shows file info
        labelText = f"{values[0]} {values[1]} {values[2]} {values[3]}"

        #label to confirm selected file
        fileInfoLabel = tk.Label(confirmDownloadWindow, text=labelText,
font=("Arial",15))

        #button for downloading files
```

```
        button = tk.Button(confirmDownloadWindow,text="DOWNLOAD" ,
anchor='center', height=2, width=20,
command=lambda:MainMenu.downloadFileSelf(values))

        fileInfoLabel.pack()

        button.pack()


    #get current item of file viewer after downlaod
    def onDoubleClick(a):

        currentItem = a.focus()

        currentItemArray = a.item(currentItem)

        #get values of array
        fileValues = currentItemArray['values']

        #create popup screen confirming the user wants to download
selected file

        MainMenu.confirmDownloadWindow(fileValues)


    def onDoubleClickShared(a):

        currentItem = a.focus()

        currentItemArray = a.item(currentItem)

        #get values of array
        fileValues = currentItemArray['values']

        #create popup screen confirming the user wants to download
selected file

        MainMenu.confirmDownloadWindowShared(fileValues)


    # Pop up window for myFiles section button when pressed
```

```
    def myFilesScreen():

        #send message to SFSA CLIENT to tell use we need to pull csv from
server

        #message for SELFDX
        message = FileManager.getCurrentUser()

        #send ti to messageQ
        SFSA_Client.addMessagetoQ("SELFDX " + message)


        #wait for row data to populate local csv

        time.sleep(5)
        myFilesWindow = tk.Toplevel()
        myFilesWindow.title('File Viewer')

        #define columns
        columns = ('n','d','t','u')

        #make a treeview
        myFilesTree = ttk.Treeview(myFilesWindow, columns=columns,
show='headings')

        #define headings
        myFilesTree.heading('n',text='Name')
        myFilesTree.heading('d',text='Date')
        myFilesTree.heading('t',text='Time')
        myFilesTree.heading('u',text='User')

        path = "SFSA_ACC/"
        username = FileManager.getCurrentUser()
        path = Path(path+username+".csv")

        with open(path) as f:
            reader = csv.DictReader(f, delimiter=',')
            for row in reader:
                name = row['filename']
                date = row['date']
                u_time = row['time']
```

```
                user = row['name']
                myFilesTree.insert("", 0, values=(name, date,
u_time,user))
            f.close()

        #bind double click to tree
        myFilesTree.bind("<Double-1>", lambda event, a=myFilesTree:
MainMenu.onDoubleClick(a))
        #pack tree
        myFilesTree.pack()

    #function to pull all shared files
    def sharedFilesScreen():
        #send message to SFSA CLIENT to tell use we need to pull csv from
server

        #message for SELFDX
        message = FileManager.getCurrentUser()

        #send ti to messageQ
        SFSA_Client.addMessagetoQ("SHARED " + message)


        #wait for row data to populate local csv

        time.sleep(5)
        myFilesWindow = tk.Toplevel()
        myFilesWindow.title('File Viewer')

        #define columns
        columns = ('n','d','t','u')

        #make a treeview
        myFilesTree = ttk.Treeview(myFilesWindow, columns=columns,
show='headings')

        #define headings
        myFilesTree.heading('n',text='Name')
        myFilesTree.heading('d',text='Date')
        myFilesTree.heading('t',text='Time')
        myFilesTree.heading('u',text='User')
```

```
        path = "SFSA_ACC/"
        username = FileManager.getCurrentUser()
        path = Path(path+username+".csv")

        with open(path) as f:
            reader = csv.DictReader(f, delimiter=',')
            for row in reader:
                name = row['filename']
                date = row['date']
                u_time = row['time']
                user = row['name']
                myFilesTree.insert("", 0, values=(name, date,
u_time,user))
            f.close()

        #bind double click to tree
        myFilesTree.bind("<Double-1>", lambda event, a=myFilesTree:
MainMenu.onDoubleClickShared(a))
        #pack tree
        myFilesTree.pack()
# ************* GROUPS POP UP SCREENS
*********************************************************************

    def displayGroups(tree):

        username = FileManager.getCurrentUser()
        path = Path(f"SFSA_ACC/{username}/Groups/masterfile.csv")
        with open(path) as f:
            reader = csv.DictReader(f, delimiter=',')
            for row in reader:
                name = row['name']
                date = row['date']
                u_time = row['time']
                host = row['host']
                tree.insert("", 0, values=(name, date, u_time,host))
            f.close()

    # Pop up window for viewing groups
    def GroupsScreen():
```

```
        groupsWindow = tk.Toplevel()
        groupsWindow.title('View Groups')
        #define columns
        columns = ('n','d','t','h')

        #make a treeview
        groupsTree = ttk.Treeview(groupsWindow, columns=columns,
show='headings')

        #define headings
        groupsTree.heading('n',text='Group Name')
        groupsTree.heading('d',text='Date Added')
        groupsTree.heading('t',text='Time Added')
        groupsTree.heading('h',text="Host")

        MainMenu.displayGroups(groupsTree)

        #pack tree
        groupsTree.pack()


    # Pop up window for group management

    def manageGroupsScreen():

        groupsWindow = tk.Toplevel()
        groupsWindow.title('Manage Groups')

        # Create frame for window
        masterFrame = tk.Frame(groupsWindow)

        masterFrame.pack()

        # Create frame for Group Management

        groupFrame = tk.Frame(masterFrame)

        # Create frame for Group Managment buttons
        groupFrameButtons = tk.Frame(groupFrame)

        # Create frame for user Management
```

```
        userFrame = tk.Frame(masterFrame)

        # Create frame for user Management Buttons
        userFrameButtons = tk.Frame(userFrame)


        #******************** GROUPS FRAME
 ****************************************

        # Label for Group Frame
        groupLabel = tk.Label(masterFrame, text="Groups")
        #define columns
        group_columns = ('n','d','t','h')

        #make a treeview
        groupsTree = ttk.Treeview(groupFrame, columns=group_columns,
show='headings')

        #define headings
        groupsTree.heading('n',text='Group Name')
        groupsTree.heading('d',text='Date Added')
        groupsTree.heading('t',text='Time Added')
        groupsTree.heading('h',text="Host")

        MainMenu.displayGroups(groupsTree)

        #buttons for group frame
        add_btn_group =   tk.Button(groupFrameButtons,text="ADD" ,
anchor='center', height=2, width=20,
command=lambda:MainMenu.addGroupFrame(groupsTree))
        remove_btn_group =   tk.Button(groupFrameButtons,text="REMOVE",
anchor='center', height=2, width=20,
command=lambda:MainMenu.removeGroupFrame(groupsTree))

        groupsTree.bind("<Double-1>", lambda event, a=groupsTree:
MainMenu.viewUsers(groupsTree,userTree))

         #pack tree
        groupLabel.pack()
        groupFrame.pack()
        groupsTree.pack(side=tk.LEFT)
```

```
        groupFrameButtons.pack(side=tk.LEFT)
        add_btn_group.pack()
        remove_btn_group.pack()

        # ******************* USER FRAME
************************************************************

        # Label for Group Frame
        userLabel = tk.Label(masterFrame, text="Users")

        #define columns
        user_columns = ('u','d','t','h')

        #make a treeview
        userTree = ttk.Treeview(userFrame, columns=user_columns,
show='headings')

        #define headings
        userTree.heading('u',text='User Name')
        userTree.heading('d',text='Date Added')
        userTree.heading('t',text='Time Added')
        userTree.heading('h',text="Host")

        #buttons for group frame
        add_btn_user =   tk.Button(userFrameButtons,text="ADD" ,
anchor='center', height=2, width=20,
command=lambda:MainMenu.addUserFrame(groupsTree,userTree))
        remove_btn_user =   tk.Button(userFrameButtons,text="REMOVE",
anchor='center', height=2, width=20,
command=lambda:MainMenu.removeUserFrame(groupsTree,userTree))

        userLabel.pack()
        userFrame.pack()
        userTree.pack(side=tk.LEFT)
        userFrameButtons.pack(side=tk.LEFT)
        add_btn_user.pack()
        remove_btn_user.pack()
```

```
# ************************** ADD and CREATE GROUP POP UPS
**************************************************************************
**********************************


    def getSelectedGroup(a):
        currentItem = a.focus()

        currentItemArray = a.item(currentItem)

        #get values of array
        fileValues = currentItemArray['values']

        group = fileValues[0].strip()

        return group



    def addUserToGroup(frame, userTree,groupframe,userEntry):

        group = MainMenu.getSelectedGroup(groupframe)

        user = userEntry.get().strip()

        GroupManager.addUserToGroupFile(FileManager.getCurrentUser(),
group,user)

        frame.destroy()

        MainMenu.displayUsers(userTree,group)

    def displayUsers(userTree,group):
       #cleartree
        for item in userTree.get_children():
            userTree.delete(item)

        username = FileManager.getCurrentUser()
        path = Path(f"SFSA_ACC/{username}/Groups/{group}.csv")
        with open(path) as f:
            reader = csv.DictReader(f, delimiter=',')
```

```
            for row in reader:
                name = row['user']
                date = row['date']
                u_time = row['time']
                host = row['host']
                userTree.insert("", 0, values=(name, date, u_time,host))
            f.close()

    def viewUsers(groupTree,userTree):
        group = MainMenu.getSelectedGroup(groupTree)
        #print("Group is ", group)
        MainMenu.displayUsers(userTree,group)



    #Add a user to group prompt
    def addUserFrame(groupsTree,userTree):
        addUserPrompt = tk.Toplevel()

        #create frame for label and buttons
        masterFrame = tk.Frame(addUserPrompt)
        buttonsFrame = tk.Frame(masterFrame)

        masterFrame.pack()

        #create label
        entryLabel = tk.Label(masterFrame, text="Enter Username: ")

        # create entry
        userEntry = tk.Entry(masterFrame)

        #create add and cancel buttons
        addButton = tk.Button(buttonsFrame,text="ADD", height=2, width=20,
command=lambda:MainMenu.addUserToGroup(addUserPrompt,userTree,groupsTree,u
serEntry))
        cancelButton= tk.Button(buttonsFrame, text= "CANCEL", height=2,
width=20, command=lambda:MainMenu.cancelGroupButton(addUserPrompt))



        #pack
```

```
        entryLabel.pack(pady=5)
        userEntry.pack(ipadx= 10, ipady=10, pady= 5)


        buttonsFrame.pack(pady=5 )

        cancelButton.pack(side=tk.LEFT)
        addButton.pack(side=tk.LEFT)

    #get selected User
    def getSelectedUser(userTree):
        currentItem = userTree.focus()

        currentItemArray = userTree.item(currentItem)

        #get values of array
        fileValues = currentItemArray['values']

        user = fileValues[0].strip()

        return user

    #remove user
    def removeUserButton(window,groupTree,userTree):

        group = MainMenu.getSelectedGroup(groupTree)
        user = MainMenu.getSelectedUser(userTree)


GroupManager.removeUserFromGroup(FileManager.getCurrentUser(),group,user)


        MainMenu.displayUsers(userTree,group)

        window.destroy()


    def removeUserFrame(groupTree,userTree):
        removeUserPrompt = tk.Toplevel()

        # Create frames and pack master for popup
```

```
        masterFrame = tk.Frame(removeUserPrompt)

        masterFrame.pack()

        # Create frame for button
        buttonsFrame = tk.Frame(masterFrame)

        #Create label for 'Are you sure?' prompt
        confirmLabel = tk.Label(masterFrame,text="Are you sure you want to
delete this user? ", font=("Arial",15))

        #Create buttons 'YES' and 'NO'

        yesButton =  tk.Button(buttonsFrame,text="YES", padx=5, height=2,
width=20,
command=lambda:MainMenu.removeUserButton(removeUserPrompt,groupTree,userTr
ee))
        noButton  =  tk.Button(buttonsFrame, text="NO", padx= 5, height=2,
width=20, command=lambda:MainMenu.cancelGroupButton(removeUserPrompt))

        # Pack contents

        confirmLabel.pack()
        buttonsFrame.pack(pady=5)

        noButton.pack(side=tk.LEFT)
        yesButton.pack(side=tk.LEFT)



    #on double click group frame

    def removeGroupFromManager(window,a):

        currentItem = a.focus()

        currentItemArray = a.item(currentItem)


        if currentItemArray:
            #get values of array
```

```
            groupValues = currentItemArray['values']

            print("current GROUP FRAME VALues are", groupValues)



            #remove from master file

GroupManager.removeFromGroupMasterFile(FileManager.getCurrentUser,groupVal
ues[0].strip(),groupValues[1].strip(),groupValues[2].strip(),groupValues[3
].strip())

            #cleartree
            for item in a.get_children():
                a.delete(item)

            #display groups
            MainMenu.displayGroups(a)

            #destroy window
            window.destroy()

    #control for the gui and calls group manager function to add group
    def addGroupButton(groupsTree,window,username,u_entry):
        #add text file
        GroupManager.addGroup(username,u_entry)
        #cleartree
        for item in groupsTree.get_children():
            groupsTree.delete(item)

        MainMenu.displayGroups(groupsTree)
        #close window
        window.destroy()

    def cancelGroupButton(window):
        #close the window
        window.destroy()

    #Add group to
    def addGroupFrame(groupsTree):
```

```
        addGroupPrompt = tk.Toplevel()

        #create frame for label and buttons
        masterFrame = tk.Frame(addGroupPrompt)
        buttonsFrame = tk.Frame(masterFrame)

        masterFrame.pack()

        #create label
        entryLabel = tk.Label(masterFrame, text="Enter Group Name: ")

        # create entry
        groupEntry = tk.Entry(masterFrame)

        #create add and cancel buttons
        addButton = tk.Button(buttonsFrame,text="ADD", height=2, width=20,
command=lambda:MainMenu.addGroupButton(groupsTree,addGroupPrompt,FileManag
er.getCurrentUser(),groupEntry.get().strip()))
        cancelButton= tk.Button(buttonsFrame, text= "CANCEL", height=2,
width=20,command=lambda:MainMenu.cancelGroupButton(addGroupPrompt))


        #pack
        entryLabel.pack(pady=5)
        groupEntry.pack(ipadx= 10, ipady=10, pady= 5)


        buttonsFrame.pack(pady=5 )

        cancelButton.pack(side=tk.LEFT)
        addButton.pack(side=tk.LEFT)

    def removeGroupFrame(groupTree):
        removeGroupPrompt = tk.Toplevel()

        # Create frames and pack master for popup
        masterFrame = tk.Frame(removeGroupPrompt)

        masterFrame.pack()

        # Create frame for button
```

```
        buttonsFrame = tk.Frame(masterFrame)

        #Create label for 'Are you sure?' prompt
        confirmLabel = tk.Label(masterFrame,text="Are you sure you want to
delete this group? ", font=("Arial",15))

        #Create buttons 'YES' and 'NO'

        yesButton =  tk.Button(buttonsFrame,text="YES", padx=5, height=2,
width=20,
command=lambda:MainMenu.removeGroupFromManager(removeGroupPrompt,groupTree
))
        noButton  =  tk.Button(buttonsFrame, text="NO", padx= 5, height=2,
width=20, command=lambda:MainMenu.cancelGroupButton(removeGroupPrompt))


        # Pack contents

        confirmLabel.pack()
        buttonsFrame.pack(pady=5)

        noButton.pack(side=tk.LEFT)
        yesButton.pack(side=tk.LEFT)



# ***************************BROWSE FILES
**********************************************************************
    #tkinter window that lets user browse files
    def browseFiles(groupWindow,group):

        groupWindow.destroy()

        share_group = group.strip()

        filename = filedialog.askopenfilename(initialdir="/",
                                              title="Select a File",
                                              filetypes=(
                                                  ("Text files", "*.txt"),
                                                  ("all files", "*")))
```

```
        if filename:

            user = FileManager.getCurrentUser()
            #append file information to users

            FileManager.encryptFile(user,filename,share_group)


        #return filename

    #upload file and update lsitbox
    def browseFilesMyFiles(listbox):
        filename = filedialog.askopenfilename(initialdir="/",
                                              title="Select a File",
                                              filetypes=(
                                                  ("Text files", "*.txt"),
                                                  ("all files", "*")))


        if filename:

            listbox.insert("end",filename)

            print(FileManager.getCurrentUser())


FileManager.encryptFile(FileManager.getCurrentUser(),filename,'self')

        else:
            return False

    #for select group function

    def show(label,clicked):
        label.config(text = clicked.get())
    #creates a popup drop down menu for groups
    def selectGroup():

        #create toplevel window
        groupWindow = tk.Toplevel()
```

```
        groupWindow.geometry("200x200")


        #get user from group
        user = FileManager.getCurrentUser()

        #array to hold group options
        groupOptions = []

        #get path of masterfile csv

        path = f"SFSA_ACC/{user}/Groups/masterfile.csv"

        #for loop to loop through master csv and display options
        with open(path) as f:
            reader = csv.DictReader(f, delimiter=',')
            for row in reader:
                name = row['name']
                groupOptions.append(name)

            f.close()
        #data type of menu text
        clicked = tk.StringVar()

        if groupOptions:
            clicked.set(groupOptions[0])
        else:
            clicked = "None"

        selectGroupLabel = tk.Label(groupWindow, text = " ")

        #create drop down menu
        dropDownMenu = tk.OptionMenu(groupWindow,clicked, *groupOptions,
command=lambda x=None: MainMenu.show(selectGroupLabel,clicked))

        uploadBtn = tk.Button(groupWindow, text="SHARE",
command=lambda:MainMenu.browseFiles(groupWindow,selectGroupLabel.cget('tex
t')))
        #creat label

        selectGroupLabel.pack()
```

```
        dropDownMenu.pack()
        uploadBtn.pack()


# ***************************PARENT FRAME
*************************************************************************
    def __init__(self,parent,controller):

        tk.Frame.__init__(self,parent)


         #set the weights of the columns and rows
        self.grid_columnconfigure(0, weight=1)
        self.grid_columnconfigure(1, weight=3)
        self.grid_columnconfigure(2, weight=1)

        self.grid_rowconfigure(0,weight=1)
        self.grid_rowconfigure(1,weight=1)
        self.grid_rowconfigure(2,weight=1)
        self.grid_rowconfigure(3,weight=1)


        # *********** MY FILES ************************************

        #frame for myfiles widgets
        myFilesframe = tk.Frame(self,
highlightbackground="#20bebe",highlightthickness=2)
        myFilesframe.grid(column=0, row=1)

        #frame for myfiles buttons
        myFilesButtons = tk.Frame(self)
        myFilesButtons.grid(column=0,row=3)

        #frame for myFiles ListBox

        myFiles_lbf = tk.Frame(self)
        myFiles_lbf.grid(column=0,row=2)

        #listbox
        myFiles_lb = tk.Listbox(myFiles_lbf, width=50, height=20)
```

```
        #scroll bar for listbox
        myfiles_scroll = tk.Scrollbar(myFiles_lbf)

        #Label for myFiles
        myFilesLabel = tk.Label(myFilesframe, text="My
Files",font=("Arial",15))

        #Image for myFiles
        myFilesImage = Image.open("my_files1.png")
        myFiles_tk = ImageTk.PhotoImage(myFilesImage)
        myFilesImageLabel = tk.Label(myFilesframe,image=myFiles_tk)
        myFilesImageLabel.image = myFiles_tk


        #Buttons For myfiles
        #myFilesUpload_btn =   tk.Button(myFilesButtons,text="UPLOAD",
bg="#20bebe", fg="white", anchor='center', height=2, width=20,
command=MainMenu.browseFiles)
        myFilesUpload_btn =   tk.Button(myFilesButtons,text="UPLOAD",
bg="#20bebe", fg="white", anchor='center', height=2, width=20, command=
lambda : MainMenu.browseFilesMyFiles(myFiles_lb))

        myFilesDownload_btn =
tk.Button(myFilesButtons,text="DOWNLOAD",bg="#20bebe", fg="white",
anchor='center', height=2, width=20, command=MainMenu.myFilesScreen )


        # Pack myFiles

        myFiles_lb.pack(side=tk.LEFT, fill=tk.BOTH)
        myfiles_scroll.pack(side=tk.RIGHT, fill=tk.Y)

        myFilesLabel.pack(pady=20,padx=20)
        myFilesImageLabel.pack(pady=20,padx=20)

        myFilesUpload_btn.pack(ipady=10,pady=20)
        myFilesDownload_btn.pack(ipady=10)


        # ********** SHARED FILES ************************************
```

```
        #frame for sharedFiles widgets
        sharedFilesFrame =
tk.Frame(self,highlightbackground="#20bebe",highlightthickness=2)
        sharedFilesFrame.grid(column=1, row=1)

        #frame for sharedFiles buttons
        sharedFilesButtons = tk.Frame(self)
        sharedFilesButtons.grid(column=1,row=3)

        #frame for shared ListBox

        sharedFiles_lbf = tk.Frame(self)
        sharedFiles_lbf.grid(column=1,row=2)

        #listbox
        sharedFiles_lb = tk.Listbox(sharedFiles_lbf, width=50, height=20)

        #scroll bar for listbox
        sharedfiles_scroll = tk.Scrollbar(sharedFiles_lbf)

        #Label for sharedFiles
        sharedFilesLabel = tk.Label(sharedFilesFrame, text="Shared
Files",font=("Arial",15))

        #Image for sharedFiles
        sharedFilesImage = Image.open("shared_files1.png")
        sharedFiles_tk = ImageTk.PhotoImage(sharedFilesImage)
        sharedFilesImageLabel =
tk.Label(sharedFilesFrame,image=sharedFiles_tk)
        sharedFilesImageLabel.image = sharedFiles_tk


        #Buttons For sharedFiles
        sharedFilesView_btn =
tk.Button(sharedFilesButtons,text="UPLOAD", bg="#20bebe", fg="white",
anchor='center', height=2, width=20, command=MainMenu.selectGroup)
        sharedFilesDownload_btn =
tk.Button(sharedFilesButtons,text="DOWNLOAD",bg="#20bebe", fg="white",
anchor='center', height=2, width=20, command=MainMenu.sharedFilesScreen)

        #Pack Shared Files
```

```
        sharedFiles_lb.pack(side=tk.LEFT, fill=tk.BOTH)
        sharedfiles_scroll.pack(side=tk.RIGHT, fill=tk.Y)

        sharedFilesLabel.pack(pady=20,padx=20)
        sharedFilesImageLabel.pack(pady=20,padx=20)

        sharedFilesView_btn.pack(ipady=10,pady=20)
        sharedFilesDownload_btn.pack(ipady=10)



        # *********** GROUPS  ************************************

        #frame for groups
        groupFrame =
tk.Frame(self,highlightbackground="#20bebe",highlightthickness=2)
        groupFrame.grid(column=2, row=1)

        #frame for groups buttons
        groupsButtons = tk.Frame(self)
        groupsButtons.grid(column=2,row=3)

        #frame for shared ListBox

        groups_lbf = tk.Frame(self)
        groups_lbf.grid(column=2,row=2)

        #listbox
        groups_lb = tk.Listbox(groups_lbf, width=50, height=20)

        #scroll bar for listbox
        groups_scroll = tk.Scrollbar(groups_lbf)

        #Label for groups
        groupsLabel = tk.Label(groupFrame,
text="Groups",font=("Arial",15))

        #Image for groups
        groupImage = Image.open("group_icon.png")
        group_tk = ImageTk.PhotoImage(groupImage)
```

```
        groupImageLabel = tk.Label(groupFrame,image=group_tk)
        groupImageLabel.image = group_tk



        #   Buttons For Groups
        groupsView_btn =   tk.Button(groupsButtons,text="VIEW",
bg="#20bebe", fg="white", anchor='center', height=2, width=20,
command=MainMenu.GroupsScreen)
        groupsCreate_btn =
tk.Button(groupsButtons,text="CREATE",bg="#20bebe", fg="white",
anchor='center', height=2, width=20, command=MainMenu.manageGroupsScreen)


        #Pack groups
        groups_lb.pack(side=tk.LEFT, fill=tk.BOTH)
        groups_scroll.pack(side=tk.RIGHT, fill=tk.Y)

        groupsLabel.pack(pady=20,padx=20)
        groupImageLabel.pack(pady=20,padx=20)

        groupsView_btn.pack(ipady=10,pady=20)
        groupsCreate_btn.pack(ipady=10)




#class to run tkinter mainloop in a thread so we can send info to
# client service
#class App(threading.Thread):
#    def __init__(self):
#        threading.Thread.__init__(self)
#        self.start()

#    def callback(self):
#        self.root.quit()

#    def run(self):
#        self.app = SFSA()
#        self.app.mainloop()
```

```
#class to run our client side of the app on a thread
#class Client(threading.Thread):
#    def __init__(self):
#        threading.Thread.__init__(self)
#        self.start()
#    def callback(self):
#        self.quit()

#    def run(self):
#        SFSA_Client.main()


#Run our app

#app = App()
#client = Client()
```

SFSA_Client.py

```
from operator import index
from pathlib import Path
import socket
from time import sleep
import Accounts
import sys
import errno
import random
from random import seed
import string
import tkinterWarningInstance
import pickle
```

```
import select
import csv
import FileManager
import pandas as pd

# CONSTANTS

IP = "10.0.2.15"
PORT = 1234
ADDR = (IP,PORT)
SIZE = 1024
FORMAT = "utf-8"
HEADER_LENGTH = 20
DISCONNECT_MSG = "!DISCONNECT"


#HEADER TYPES
#Headers attacthed to the beginning of the message to be handled by the
client
H_SIGN_UP = "SIGNUP"
H_LOGIN = "LOGINX"

#seed for random user id generation

# Create Queue of messages


byte_type = ""
queue = []

#byte type

def setByteType(m_type):
    global byte_type
    byte_type = m_type

def getByteType():
    return byte_type

# add strings to queue to send over
def addMessagetoQ(message):
```

```
   queue.append(message)

# remove message from queue to send over to server
def removeMessageFromQ(q,i):
   q.pop(i)

# generate Client ID
# Identifies the client as a 4 Digit 2 Letter combonation
# Example is 1234AA is one user id
# This user id is going to be stored in a dictonary and the ip and socket
the client sends will get the user id
def generateClientID():

    #get all ascii letters
    lower_upper_alphabet = string.ascii_letters

    # Generate random int values
    value1 = random.randint(0,9)
    value2 = random.randint(0,9)
    value3 = random.randint(0,9)
    value4 = random.randint(0,9)
    letter1 = random.choice(lower_upper_alphabet)
    letter2 = random.choice(lower_upper_alphabet)
    #concatinated results
    clientID = str(value1) + str(value2) + str(value3) + str(value4) +
letter1 + letter2
    print(clientID)
    return clientID

#function that receives messages from host

def receive_message_from_host(client_socket):
    try:
        #get the header of the message
        message_header = client_socket.recv(HEADER_LENGTH)

        print("Mess Header: ", int(message_header.decode("utf-8")))

        if not len(message_header):
            print("not header")
```

```
            return False

        #get type of data

        m_type = client_socket.recv(6)

        if not len(m_type):
            print("not m_type")
            return False

        #strip message header
        message_length = int(message_header.decode("utf-8").strip())

        return {"header": message_header, "m_type":m_type, "data":
client_socket.recv(message_length)}

    except:
        return False

def main():

   # generate clientID
   my_clientID = generateClientID()
   #generate type
   my_type = "CONNCT"

   # Create Socket
   # Create a socket
   # socket.AF_INET - address family, IPv4, some otehr possible are
AF_INET6, AF_BLUETOOTH, AF_UNIX
   # socket.SOCK_STREAM - TCP, conection-based, socket.SOCK_DGRAM - UDP,
connectionless, datagrams, socket.SOCK_RAW - raw IP packets
   client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

   # Connect to a given ip and port
   client_socket.connect((IP, PORT))

   # Set connection to non-blocking state, so .recv() call won;t block,
just return some exception we'll handle
   client_socket.setblocking(False)
```

```
sockets_list = [client_socket]
# send ClientID to Server
clientID = my_clientID.encode('utf-8')
m_type  = my_type.encode('utf-8')
client_header = f"{len(clientID):<{HEADER_LENGTH}}".encode('utf-8')
client_socket.send(client_header + m_type + clientID)
message = ""
while True:

# Wait for queue to be > 0
 m_type = "TEST"

 if queue:
     for i in range(len(queue)):
         # set message to first item in queue
         #print("message type is ", type(message))
         #print("message type is ",type(message) == str)

         if len(queue) == 1:
            i = 0

         #print("I is ", i )
         if type(queue[i]) == str:
             message = bytes(queue[i],'utf-8')

         # If message is not empty - send it and type of string
         if message and type(queue[i]) == str:


             #Get Type Of message to send to client
             messageType = message.split()

             #check to see if we have extra data after split
             #for sign in and create account choices
             if len(messageType) > 1:


                 message = messageType[1].strip()
                 m_type = messageType[0].strip().decode('utf-8')
```

```
                #else it is a pull request from the server
                #no extra data from split

                print("TYPE IS ", m_type)

                if m_type == "SHARED":
                    print("Getting Shared Files")

                #if we are requesting to download a file that is shared
with us privately
                if m_type == "SELFDX":
                    print("sending selfdx message")



                #if we are creating a account
                if m_type == "SIGNUP":
                    message  = messageType[1].strip().decode('utf-8') + "
" + messageType[2].strip().decode('utf-8')
                    message  = message.encode('utf-8')

                #if client wants to login
                if m_type == "LOGINX":

                    message  = messageType[1].strip().decode('utf-8') + "
" + messageType[2].strip().decode('utf-8')
                    message  = message.encode('utf-8')
                # Encode message to bytes, prepare header and convert to
bytes,
                en_type = bytes(m_type,'utf-8')
                print(len(m_type))

                message_header =
f"{len(message):<{HEADER_LENGTH}}".encode('utf-8')
                client_socket.send(message_header + en_type + message)

                #remove the message from the queue
                if queue:
                    removeMessageFromQ(queue,i)
                #print("PASS: %i\n",i)
```

```
            #if message is not empty and type of bytes we are using pickle
so we cant concatnate with strings
            if queue:
                if message and type(queue[i]) == bytes:
                    # this tells the server we are sending bytes

                    message = queue[i]

                    bt_type = getByteType()

                    if bt_type == "UPLOAD_SYM_KEY":

                        m_type = "SYMKEY"

                        en_type = bytes(m_type,'utf-8')
                        print(len(m_type))

                        message_header =
f"{len(message):<{HEADER_LENGTH}}".encode('utf-8')

                        client_socket.send(bytes(message_header) +
bytes(en_type) + message)


                    if bt_type == "APPEND_USER_DOWNLOAD":

                        m_type = "BYTESX"

                        en_type = bytes(m_type,'utf-8')
                        print(len(m_type))

                        message_header =
f"{len(message):<{HEADER_LENGTH}}".encode('utf-8')

                        client_socket.send(bytes(message_header) +
bytes(en_type) + message)

                    if bt_type == "REQUEST_SYMKEY":

                        m_type = "RESYMX"
```

```
                        en_type = bytes(m_type,'utf-8')
                        print(len(m_type))

                        message_header =
f"{len(message):<{HEADER_LENGTH}}".encode('utf-8')

                        client_socket.send(bytes(message_header) +
bytes(en_type) + message)

                    if queue:
                        removeMessageFromQ(queue,i)
    try:
        # Now we want to loop over received messages (there might be more
than one) and print them
        while True:


            #call select



            # Receive our "header" containing username length, it's size
is defined and constant
            message_header = client_socket.recv(HEADER_LENGTH)

            #print(message_header.decode('utf-8'))

            # If we received no data, server gracefully closed a
connection, for example using socket.close() or
socket.shutdown(socket.SHUT_RDWR)
            if not len(message_header):
                print('Connection closed by the server')
                sys.exit()


            message_type = client_socket.recv(6).decode('utf-8')


             # If we received no data, server gracefully closed a
connection, for example using socket.close() or
socket.shutdown(socket.SHUT_RDWR)
```

```
            if not len(message_type):
                print('Connection closed by the server')
                sys.exit()

            # Convert header to int value
            message_length = int(message_header.decode('utf-8').strip())

            if message_type == "SIGNUP" or message_type == "LOGINX":

                message =
client_socket.recv(message_length).decode('utf-8')


            #interact with GUI

            #SIGNUP CASES
            if(message == "AVAILABLE"):
                tkinterWarningInstance.accountMessages(message)
            elif(message == "TAKEN"):
                tkinterWarningInstance.accountMessages(message)

            #LOGIN CASES

            if(message == "GOOD_LOGIN"):
                #print("GOOD")
                #print("message:",message, " ", message_length)
                tkinterWarningInstance.switchScreen(message)

            elif(message == "BAD_LOGIN"):
                tkinterWarningInstance.badLogin()



            # MY FILES CASES


            #Print message
            #print(f'{message}')
```

```
            #unpickle any files we receive

            if message_type == "SELFDX":
                message = client_socket.recv(2048)

                rows_unpickled = pickle.loads(message)

                path = "SFSA_ACC/"
                username = FileManager.getCurrentUser()
                path = Path(path+username+".csv")

                with open(path,'w',newline='') as csvfile:

                    csvwriter = csv.writer(csvfile)

                    for row in rows_unpickled:


                        csvwriter.writerow(row)

                    csvfile.close()
                #print("Rows unpickled is ", rows_unpickled)

            if message_type == "SHARED":
                message = client_socket.recv(2048)

                rows_unpickled = pickle.loads(message)

                path = "SFSA_ACC/"
                username = FileManager.getCurrentUser()
                path = Path(path+username+".csv")

                with open(path,'w',newline='') as csvfile1:
                    csvwriter = csv.writer(csvfile1)
                    for row in rows_unpickled:
                        csvwriter.writerow(row)
                    csvfile1.close()


                #filter out csv with only different group names
                df = pd.read_csv(f"SFSA_ACC/{username}.csv")
```

```
            differntGroupDF = df.loc[ (df['group'] != "self") ]
            differntGroupDF.to_csv(path, index=False)

            #print("Rows unpickled is ", rows_unpickled)
        if message_type == "KEYSXX":

            message = client_socket.recv(4096)

            keys_unpickled = pickle.loads(message)

            username = FileManager.getCurrentUser()
            #place keys
            FileManager.placeKeys(username,keys_unpickled)


        #download symetric key
        if message_type == "RESYMX":
            message = client_socket.recv(2048)

            sym_unpickled = pickle.loads(message)

            filename = sym_unpickled[0]

            sym_key_bytes = sym_unpickled[1]

            path = "SFSA_ACC/"

            username = FileManager.getCurrentUser()

            path = f"{path}{username}/encryptedKeyStore/"

            #create symkey
            with open(path+f"{filename}encrypted.key","wb") as f:
                f.write(sym_key_bytes)
                f.close()

        # MY FILES CASES


        #Print message
        #print(f'{message}')
```

```
    except IOError as e:
        # This is normal on non blocking connections - when there are no
incoming data error is going to be raised
        # Some operating systems will indicate that using AGAIN, and some
using WOULDBLOCK error code
        # We are going to check for both - if one of them - that's
expected, means no incoming data, continue as normal
        # If we got different error code - something happened
        if e.errno != errno.EAGAIN and e.errno != errno.EWOULDBLOCK:
            print('Reading error: {}'.format(str(e)))
            sys.exit()

        # We just did not receive anything
        continue

    #except Exception as e:
    #    # Any other exception - something happened, exit
    #    print('Reading error Exception: '.format(str(e)))
    #    sys.exit()



if __name__ == "__main__":
    main()
```

SFSA_runner.py

```
import threading
import SFSA
import SFSA_Client
import ipfsApi

#class to run tkinter mainloop in a thread so we can send info to
# client service
```

```
class App(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.start()

    def callback(self):
        self.root.quit()

    def run(self):
        self.app = SFSA.SFSA()
        self.app.mainloop()

#class to run our client side of the app on a thread
class Client(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.start()
    def callback(self):
        self.quit()

    def run(self):
        SFSA_Client.main()


#Run our app

app = App()
client = Client()
```

tkinterWarningInstance.py

```
#create seperate tkinter instances for warning boxes

from stat import SF_APPEND
import tkinter as tk
```

```
import SFSA
def accountMessages(message):
        if message == "AVAILABLE":
           tk.messagebox.showinfo(message="Account Created Successfully")
        elif message  == "TAKEN":
            tk.messagebox.showerror(message="Username TAKEN Please enter
anonther")
def badLogin():
            tk.messagebox.showerror(message="BAD LOGIN")
def switchScreen(message):
            if message == "GOOD_LOGIN":
               SFSA.LoginScreen.switchFrame = True
```

AccountStorage.py

```
# AccountStorage.py
# Preston Robbins
# 2/10/22
# This file is reponsible for adding usernames and passwords that are sent
from the client server
# to the servers local file storage via a text file
# This program will read and send a message to the server if the username
is taken
# This program will make sure logins are valid and send a message back to
the server'



# check username
# This function checks if a username requested is available
def check_username_av(username):
    f = open('database.txt','r')
    lines = f.readlines()
    for i in lines:
        userLine  = i.split()
        fileUserName  = userLine[0].strip()

        if username == fileUserName:
            f.close()
```

```
            return False
    f.close()
    return True


    print()

# confirm login
# This function confirms that the user entered the correct information at
the login screen

def verify_login_info(username,password):
    f = open('database.txt','r')
    lines = f.readlines()
    for i in lines:
        userLine  = i.split()
        fileUserName  = userLine[0].strip()
        filePassword   = userLine[1].strip()
        print("username:%s password:%s" %(username,password))

        if username == fileUserName and password == filePassword:
            f.close()
            return True
    f.close()
    return False
# add account
# This function adds a account to the file when the user passes
check_username

def add_account(username,password):
    f = open("database.txt","a")
    userString = username + " " + password
    if(check_username_av(username) == True):
        f.write("\n")
        f.write(userString)
    f.close()


#add_account("Super01","Comic@14")
#print(check_username_av("Superpositive01"))
#print(verify_login_info("Superpositive01","Koolaid"))
```

```
#file1 = open('database.txt', 'r')
#print(file1.read())
#file1.close()
```

Filemanagerserver.py

```
# This file manages files amongst the server

from asyncore import read
from cmath import pi
from pathlib import Path
import csv
from tabnanny import check
from cryptography.fernet import Fernet
import os
import ipfsApi
import time
from datetime import date, datetime
import rsa

#header for csv
header = ['name','filename','hash','date','time','group']

#path
path="SFSA_MAIN/"

#check for file in local directory that is u_name.txt
#do this on login
def checkForFile(name):
    filename = name.strip()
    filepath = f"SFSA_MAIN/{filename}"
    filepath = Path(filepath)
    print(filepath)
    print(filepath.is_file())
```

```
    return filepath.is_file()

#create the file for login if it does not exsist
def createUserNameFile(name):
    filename = name +".csv"
    filepath = Path(path+filename)
    print("File Path is " , filepath)
    if checkForFile(filename) == False:
        with open(filepath, 'w', encoding='UTF8') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow(header)

            csvfile.close()

#create groupFolder
def createGroupFolder(name):
    #file path to folder
    folderpath = Path(f"SFSA_MAIN/Groups/{name}/")

    if os.path.exists(folderpath) == False:
        os.mkdir(folderpath)
    else:
        print("GOOD PATH FOR")



#check if keys for user already exsist
def checkKeys(name):
    privateKey = name + "private" + ".key"
    publicKey  = name + "public"  + ".key"
    symetricKey = name + "symetric" + ".key"

    keyPath = f"SFSA_MAIN/{name}/"

    privateKeyPath = Path(keyPath+privateKey)
    publicKeyPath = Path(keyPath+publicKey)
    symetricKeyPath = Path(keyPath+symetricKey)
```

```
    if (privateKeyPath.is_file() and publicKeyPath.is_file() and
symetricKeyPath.is_file()) == True:
        print("Keys Are GOOD!")
        return True

    else:
        print("Keys FALSE")
        return False

#function to create keys for user

def createKeys(name):



    #create account key directory


    if checkKeys(name) == False:

        keyPath = f"SFSA_MAIN/{name}/"
        keypathos = os.path.join(path,name)

        os.mkdir(keypathos)

        encyptedKeyPath = f"SFSA_MAIN/{name}/encryptedKeyStore/"


        os.makedirs(Path(encyptedKeyPath))

        #generate symetric key
        key = Fernet.generate_key()
        k = open(keyPath+f'{name}symetric.key','wb')
        k.write(key)
        k.close()

        #create the pub and private key
        (pubkey,privkey) = rsa.newkeys(2048)
        # generate public and private keys
        pukey = open(keyPath+f'{name}public.key','wb')
        pukey.write(pubkey.save_pkcs1('PEM'))
```

```
        pukey.close()

        prkey = open(keyPath+f'{name}private.key','wb')
        prkey.write(privkey.save_pkcs1('PEM'))
        prkey.close()
        print("CREATED KEYS")

# send keys  to user

#function to read file
def readFile(file):
    with open(file,'r') as file:
        data = file.read()
        file.close()
    return data


def sendKeys(name):

    keyArray = [None] * 3

    #open and read private key
    path = f"SFSA_MAIN/{name}/"

    #path to keys made by server
    privateKeyPath = Path(f"{path}{name}private.key")
    publicKeyPath = Path(f"{path}{name}public.key")
    symetricKeyPath = Path(f"{path}{name}symetric.key")

    #open keys and copy them
    privateKey = readFile(privateKeyPath)
    publicKey = readFile(publicKeyPath)
    symetricKey = readFile(symetricKeyPath)

    #put them in arrray
    keyArray[0] = privateKey
    keyArray[1] = publicKey
    keyArray[2] = symetricKey

    #print(keyArray)
```

```
    return keyArray
```

SFSA_SERVER.py

```
from calendar import c
from http import client
import socket
import select
import AccountStorage
import filemanagerserver
import pickle
import codecs
from pathlib import Path
import csv

#Constants for server
HEADER_LENGTH = 20
IP = "10.0.2.15"
PORT = 1234


server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#allows clients to reconnect
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

#bind IP and PORT to socket
server_socket.bind((IP,PORT))

#Listen to traffic
server_socket.listen()


#List of sockets
sockets_list = [server_socket]
```

```
#client socket key
#user data value
clients = {}

client_number = 0

#function to receive_message from client

def receive_message(client_socket):
    try:
        # get the header of the message
        message_header = client_socket.recv(HEADER_LENGTH)


        print("Mess Header: ", int(message_header.decode("utf-8")))
        #if message is not of header_length return false
        if not len(message_header):
            print("not header")
            return False


        #get the header of the message type
        m_type = client_socket.recv(6)

         #if type is not appropiate length return false
        if not len(m_type):
            print("not m_type")
            return False


         #strip message header
        message_length = int(message_header.decode("utf-8").strip())
        #strip message type


        return {"header": message_header, "m_type": m_type, "data":
client_socket.recv(message_length)}

    except:
        return False
```

```
#This will run forever until we close the server
while True:
    read_sockets, _, exception_sockets = select.select(sockets_list, [] ,
sockets_list )

    for notified_socket in read_sockets:

        # Getting a new connection
        if notified_socket == server_socket:
            #Accept the connection
            client_socket, client_address = server_socket.accept()

            #Accept sent client ID and add it to our dictonary
            clientID = receive_message(client_socket)

            #User D/C before ID is sent
            if clientID is False:
                continue
            #Add socket to list
            sockets_list.append(client_socket)

            #Add the client with client number to dictionary
            clients[client_socket] = clientID

            #print that we have a new connection
            print(f"Accpeted new connection from
{client_address[0]}:{client_address[1]}")

        # Receving a message
        else:
            message = receive_message(notified_socket)




            if message is False:
                print(f"Closed connection from {clients[client_socket]}")
```

```
                sockets_list.remove(notified_socket)
                del clients[notified_socket]
                continue

            #print(f'The message is {message["data"].decode("utf-8")}')
            print(f'The type of message is
{message["m_type"].decode("utf-8")}')

            typeOfMessage  = message["m_type"].decode("utf-8")

            #check to see if we are uploading files

           #check the data type of message if it is bytes
            #if type(message["data"]) == bytes:
                #message = message["data"]
            #else:
            if typeOfMessage != "BYTESX" and typeOfMessage != "SYMKEY" and
typeOfMessage != "RESYMX":
                message = message["data"].decode("utf-8")
            # Get user in notifed socket so we know which client sent the
message

            clientID = clients[notified_socket]
            print(f'Got message from {clientID["data"].decode("utf-8")}')

            #iterate over conneceted clients and broadcast back message

            for client_socket in clients:

                #send message back to sender of the message
                if client_socket == notified_socket:
                    print(typeOfMessage)
                    if (typeOfMessage  == "SIGNUP"):

                        #split username and password
                        messageArr = message.split()


                        if(AccountStorage.check_username_av(messageArr[0])
== True):
```

```
                            client_message = "AVAILABLE".encode('utf-8')

                            message_header =
f"{len(client_message):<{HEADER_LENGTH}}".encode('utf-8')

                            m_type = "SIGNUP".encode('utf-8')

                            client_socket.send((message_header  + m_type +
client_message))

                            #add account

AccountStorage.add_account(messageArr[0],messageArr[1])


elif(AccountStorage.check_username_av(messageArr[0]) == False):
                            client_message = "TAKEN".encode('utf-8')

                            message_header =
f"{len(client_message):<{HEADER_LENGTH}}".encode('utf-8')

                            m_type = "SIGNUP".encode('utf-8')

                            client_socket.send((message_header  + m_type +
client_message))

                    elif(typeOfMessage == "LOGINX"):

                           messageArr = message.split()
                           print(messageArr)

 if(AccountStorage.verify_login_info(messageArr[0].strip(),messageArr[1].s
trip()) == True):
                                client_message =
"GOOD_LOGIN".encode('utf-8')

                                message_header =
f"{len(client_message):<{HEADER_LENGTH}}".encode('utf-8')

                                m_type = "LOGINX".encode('utf-8')
```

```
                                client_socket.send((message_header  +
m_type + client_message))

                                #make sure master csv file is ready for
adding files

 filemanagerserver.createUserNameFile(messageArr[0].strip())

 filemanagerserver.createGroupFolder(messageArr[0].strip())
                                #create keys

 filemanagerserver.createKeys(messageArr[0].strip())

                                #send keys to user
                                keysToSend =
filemanagerserver.sendKeys(messageArr[0].strip())

                                #use pickle to send keyArray
                                pickledKeys = pickle.dumps(keysToSend)

                                #print("Pickled keys is ", pickledKeys)

                                message_header =
f"{len(client_message):<{HEADER_LENGTH}}".encode('utf-8')

                                m_type = "KEYSXX".encode('utf-8')

                                client_socket.send((bytes(message_header)
 + bytes(m_type) + pickledKeys))

                           else:
                                client_message =
"BAD_LOGIN".encode('utf-8')

                                message_header =
f"{len(client_message):<{HEADER_LENGTH}}".encode('utf-8')


                                m_type = "LOGINX".encode('utf-8')
```

```
                                client_socket.send((message_header  +
m_type + client_message))



                    elif(typeOfMessage == "SYMKEY"):
                        #load pickle data
                        data = pickle.loads(message["data"])
                        #get username
                        username = data[0]
                        #get symkey
                        sym_key = data[1]

                        file_name = data[2]


                       # print("USERNAME IS ", username)
                        #print("sym_key is ", sym_key)
                        #print("filename is ",file_name)

                        #upload symetric key
                        path = f"SFSA_MAIN/{username}/encryptedKeyStore/"
                        #f = open("demofile2.txt", "a")
                        #f.write("Now the file has more content!")
                        #f.close()
                        key =
open(Path(path+f"{file_name}encrypted.key"),"wb")
                        key.write(sym_key)
                        key.close()


                    #if we got a file uploaded
                    elif(typeOfMessage == "BYTESX"):

                        # unpickle array we got sent
                        data = pickle.loads(message["data"])

                        # check thet type of data
                        data_type = type(data)
```

```
                    print("Data Type is ", data_type)

                    # check if data is list so we can append it to
user csv
                    if data_type == list:
                        path = "SFSA_MAIN/"
                        # check for csv to add to

                        #check which group the list belongs to
                        group = data[5]
                        #first row of list is the user
                        username = data[0]

                        if group == 'self':
                            path = Path(path + username + ".csv")
                            # add row to user csv

                            with open(path,'a') as csvfile:
                                #create csv writer object
                                csvwriter = csv.writer(csvfile)

                                #write row
                                csvwriter.writerow(data)

                                csvfile.close()

                elif(typeOfMessage == "SELFDX"):


                        username = message

                        #open csv in SFSA_MAIN
                        path = "SFSA_MAIN/"

                        path = Path(path + username + ".csv")

                        #rows array for each line in text file
                        rows = []
```

```
                            with open(path,'r') as csvfile:
                                reader = csv.reader(csvfile)
                                for row in reader:
                                    rows.append(row)
                                csvfile.close()
                            #send usercsv file python

                            #print("ROWS is ", rows)

                            #pickle rows array
                            csv_pickled = pickle.dumps(rows)

                            message_header =
f"{len(client_message):<{HEADER_LENGTH}}".encode('utf-8')

                            m_type = "SELFDX".encode('utf-8')

                            #send pickled data over the wire



                            client_socket.send((bytes(message_header) +
bytes(m_type) + csv_pickled))

                        #send symetric key to user after requesting download
                        elif(typeOfMessage == "RESYMX"):
                            data = pickle.loads(message['data'])

                            username = data[0]

                            filename = data[1]

                            path =
f"SFSA_MAIN/{username}/encryptedKeyStore/{filename}encrypted.key"

                            with open(path, "rb") as f:

                                data = f.read()

                                userArray = [filename,data]
```

```
                            userArray = pickle.dumps(userArray)

                            message_header =
f"{len(client_message):<{HEADER_LENGTH}}".encode('utf-8')

                            m_type = "RESYMX".encode('utf-8')

                            client_socket.send((bytes(message_header) +
bytes(m_type) + userArray))

                            f.close()




 # It's not really necessary to have this, but will handle some socket
exceptions just in case
    for notified_socket in exception_sockets:

        # Remove from list for socket.socket()
        sockets_list.remove(notified_socket)

        # Remove from our list of users
        del clients[notified_socket]
```