FSS

File Saving Service

Full Documentation

Version 1.0

Table of Contents

Introduction

The FSS System & Protocol is designed to be a file uploading and downloading application, which is based on a server-client communication model. The concept is based upon the idea of creating an efficient file saving service (meaning uploading files to a cloud and saving those files on a remote server, securing these files until the client will request them back), based on a pretty simple communication protocol (which will be easy to implement) and an efficient data management system.

The client will send the file to the server, along with a password whose purpose is to encrypt the file later down the line. After that, he will receive from the server a cookie (an identifier for the location of the stored file) and will be used later to tell the server which of the files the client is interested to download back.

When the client is requesting to download his file back, he will send to the server the cookie he was given to, along with the password used previously to encrypt the file. Then the server will decrypt the file and send it back to the client (if the password and the cookie are correct).

In the current version (1.0) there is no support for uploading an entire directory to the server (but can be easily implemented via ZIP compression algorithm) and no user sign up support (meaning that everyone can upload a file anonymously, and only they can access the file, privately).

The FSS protocol and System can be developed in many ways to become a multi-functional product. The current version (1.0) supplies the foundations for future development.

thank you for your time, and I hope you found this product worthy to support.

To continue →

The FSS Protocol

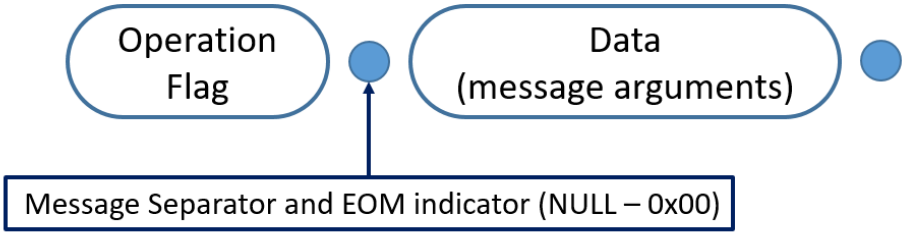
(Server-Client Communication Model)

General Information:

Query	**FSS Protocol**
Layer (ISO 5 Layer Model)	Application Layer
Stateless or Statefull	Statefull
Communication Model	Client-Server Communication
Port	954
Transported By…	TCP Protocol
Socket Security Status	Secured
Socket Security Type	Public Key Exchange – asymmetric Encryption (RSA, PKCS1_OAEP)

The FSS protocol relates to the Application Layer in the ISO 5-Layers Model. It is a Statefull protocol (user-states or levels), which is based on client-server communication. The messages are transferred on top the Transmission Control Protocol (TCP), the created socket will be secured by the RSA encryption method. The listening port (of the server) for this protocol is port 954.

Client Message Types:

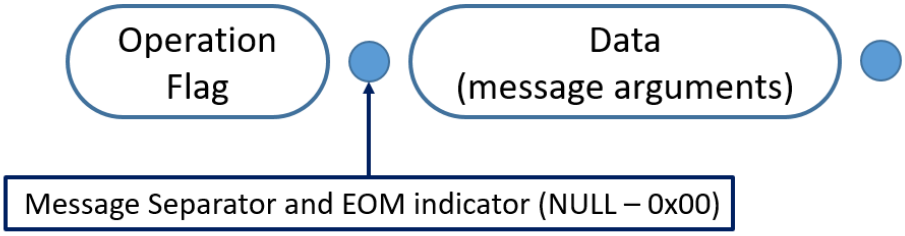


The client message structure is built like this:

- First, you have the **Operation Flag** – which indicates what type of operation the client is interested to perform. There are four types of flags:
  Upload (signed as '**U**') – for uploading file data to the server.
  Download (signed as '**D**') – for downloading file data from the server.
  File Data (signed as '**F**') – for transferring file data.
  Error (signed as '**E**') – error message type.
- Secondly, you have the **Data** sector – which contains the data for use in the operation. In the data sector the client can send:
  Password (for 'U' and 'D' flags) – for encrypting and decrypting the file.
  Cookie (only for 'D' flag) – for receiving the requested file.
  File Data (only for 'F' flag) – contains the file data encoded in base64.
  Error Details – contains information about the cause of the error.
- Finally, the **Message Separator and EOM Indicator** – which separates between the operation flag and the data sector (and inside the data sector – between arguments), also appearing at the end of the request. The Separator for our protocol is the **NULL** sign (0x00 in hex).

The file will be transported in segments in the size of the socket window. While the file transported from the server to the client, the client sends an 'OK' message, to tell the server that the file data was transported successfully as well as 'FIN' message to finish file transmission.

Server Message Types:



The server message structure is built like this:

- First, you have the **Operation Flag** – which indicates what type of data transferred from the server to the client. There are four types of flags:
  Import (signed as '**R**') – for receiving file data from the client.
  Export (signed as '**S**') – for sending file data to the client.
  File Data (signed as '**F**') – for transferring file data.
  Error (signed as '**E**') – error message type.
- Secondly, you have the **Data** sector – which contains the data intended to the client. In the data sector the client can send:
  Messages (for 'R' and 'S' flags) – for approvals (in 'S' type, password approval, in 'R' type, for file data imported successfully).
  Cookie (only for 'R' flag) – an identifier for the received file, which is sent to the client.
  File Data (only for 'F' flag) – contains the file data encoded in base64.
  Error Details – contains information about the cause of the error.
- Finally, the **Message Separator and EOM Indicator** – which separates between the operation flag and the data sector (and inside the data sector – between arguments), also appearing at the end of the message. The Separator for our protocol is the **NULL** sign (0x00 in hex).

While the file transported from the client to the server, the server sends an 'OK' message, to tell the client that the file data was imported successfully as well as 'FIN' message to finish file transmission. The server sends an 'Approved' message after password approval as well.

Client-Side Errors:

- **Wrong Password** – if the entrance code to the program is not correct, this error will be returned by the client program.
- **SQL Server cannot create a new database** – in case the SQLite service cannot create a new database for the client program, this error will be returned by the program.
- **Failed to proceed the selected operation** – if an exception was thrown by the client program or the server, which forced to close down the connection, this error will be returned by the program.
- **Record not found** – in case the requested record does not exist in the local database, this error will be returned by the client.
- **Invalid server response** – in case the message type which the server sent to the client is unrecognized, this error is returned by the client.
- **Server error** – when a server-side error is sent with details to the client.
- **Server Connection Was Failed** – when the client cannot establish a connection with the server.
- **This Operation Does Not Exist** – when the user requested a non-existing operation from the menu.

Server-Side Errors:

- **Server operation was aborted** – failure when restarting the server. Specifically, at the key-exchange stage or server reset (listening port is unavailable, binding failure ext.).
- **Illegal user state** – in case the client tries to send invalid requests.
- **Invalid cookie** – if the client sent a wrong cookie to the server.
- **Invalid password** – in case the password given by the client cannot be matched to the saved hash file (if the password is wrong).
- **cannot upload/download the file** – in case the password given by the client cannot be matched to the saved hash file (if the password is wrong).
- **Failed to proceed with the selected operation** – in case the client requested a non-existing operation.

Over Look:

As we can see, the protocol is made of stages:

First stage: sending the upload request – client view:

Client message = '**U**\x00**PASSWORD**\x00'

Server message = '**R**\x00**COOKIE**\x00'

From there, the file stream will be transported from the client to the server, until we reach the end of the file (as a result, sending a 'FIN' message'):

Client message = '**F**\x00**FILE-DATA**\x00'

Server message = '**R**\x00**OK**\x00'

…

Client message = '**F**\x00**FIN**\x00'

Server message = '**R**\x00**OK**\x00'

In this stage, the file was successfully stored on the server, and now the client can ask the file back:

Client message = '**D**\x00**COOKIE**\x00**PASSWORD**\x00'

Server message = '**S**\x00**Approved**\x00'

If the client request is valid, the file transaction stage will start:

Server message = '**F**\x00**FILE-DATA**\x00'

Client message = '**D**\x00**OK**\x00'

…

Server message = '**F**\x00**FIN**\x00'

Client message = '**D**\x00**OK**\x00'

In case of an error – the message will be: '**E**\x00**DETAILS**\x00'.

The FSS System

(Databases, file and data management)

General Information:

Query	**FSS System**
Server Database	FSS Global Database
Client Database	FSS Local Database
Database Format	SQLite3 Database
Database Management	SQLite3 Server (python API)
Databases Security Status	Not secured
File Encryption	Symmetric key encryption (AES)
Any personal information stored on the server	None (no such information included).

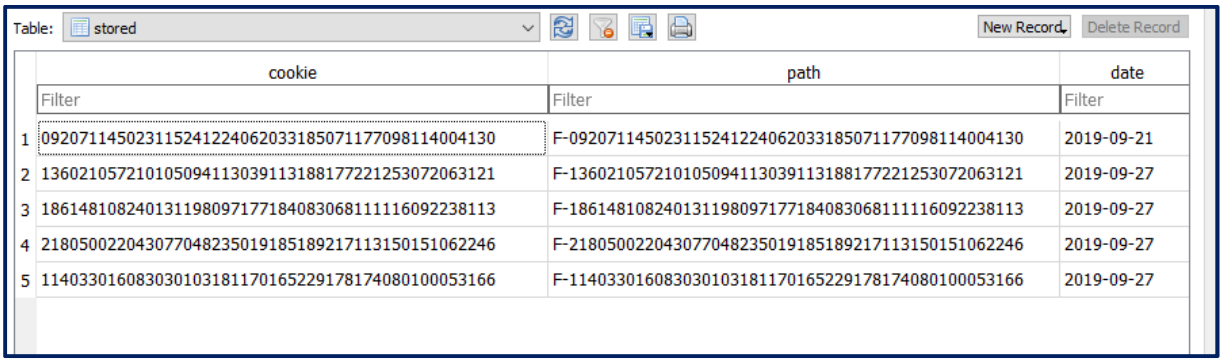
In the FSS file system, after a successful transaction of the file data from the client to the server, a record is formed on the global database. The file itself is encrypted by a given key (which the client sent to the server), via the AES encryption algorithm.

The password is stored as a hash file encoded in a SHA384 hashing algorithm.

Server-Side Database:

The global database is storing the **path** of a directory, in which the file was saved, among a generated **cookie** (file identifier) and **date** of saving.

There is only one table which consists of the saved records, labeled as '**stored**'.



	cookie	path	date
	Filter	Filter	Filter
1	092071145023115241224062033185071177098114004130	F-092071145023115241224062033185071177098114004130	2019-09-21
2	136021057210105094113039113188177221253072063121	F-136021057210105094113039113188177221253072063121	2019-09-27
3	186148108240131198097177184083068111116092238113	F-186148108240131198097177184083068111116092238113	2019-09-27
4	218050022043077048235019185189217113150151062246	F-218050022043077048235019185189217113150151062246	2019-09-27
5	114033016083030103181170165229178174080100053166	F-114033016083030103181170165229178174080100053166	2019-09-27

The first field is the cookie field. In this field, the generated cookie (which is sent to the client) used as the file identifier (a unique signature string).

The second field is the path of the file directory.

The third field is the date of saving (the day in which the file was stored on the server).

The database is managed by a few simple functions which are connecting between the FSS server and the SQLite3 server.

The functions are:

- **Create** – the function is creating the new database file for the server. The function is also adding a new table in which the data is stored.
- **Insert** – inserts a new record to the database.
- **Remove** – removes a record from the database by his cookie.
- **Clean** – deletes all record from the database.
- **Path** – searching a file by its cookie in the database and returns the directory path (in which the file is stored).

Client-Side Database:

The local database is consisting of only one table (labeled as '**cookies**'), in which the records are written. The database is storing cookies, file names, original file sizes and time of saving.

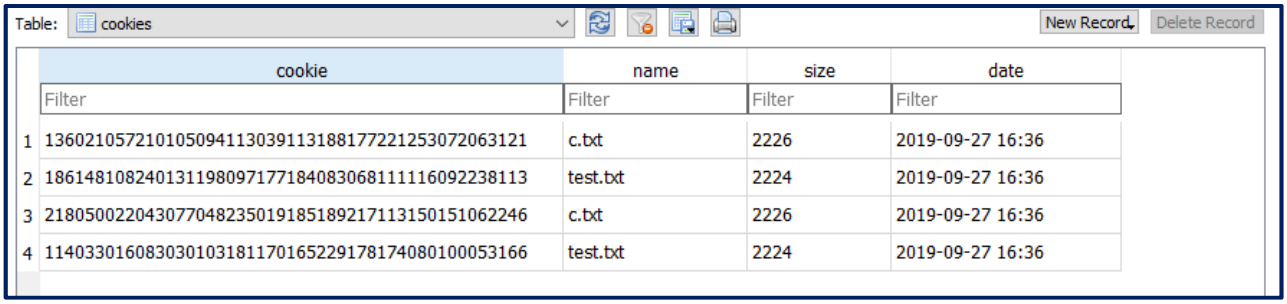


Table: cookies

New Record | Delete Record

	cookie	name	size	date
	Filter	Filter	Filter	Filter
1	13602105721010509411303911318817722125307206312l	c.txt	2226	2019-09-27 16:36
2	18614810824013119809717718408306811111609223811 3	test.txt	2224	2019-09-27 16:36
3	21805002204307704823501918518921711315015106224 6	c.txt	2226	2019-09-27 16:36
4	11403301608303010318117016522917817408010005316 6	test.txt	2224	2019-09-27 16:36

The first field ('**cookie**') is storing the file identifiers given by the server. In the future, these cookies will be used to download the files back.

The second field ('**name**') is storing the original file path of the stored file.

The third field ('**size**') is storing the original size of the file. In the future, this parameter will be used to remove the padding from the decryption process.

The last field ('**date**') consists of the time when the file was stored on the server.

The database is managed by a few simple functions which are connecting between the FSS client and the SQLite3 server.

The functions are:

- **Create** – the function is creating the new database file for the server. The function is also adding a new table in which the data is stored.
- **Insert** – inserts a new record to the database.
- **Remove** – removes a record from the database by his cookie.
- **Clean** – deletes all record from the database.
- **Search** – searching a file by its name in the local database.
- **List Files** – lists all saved files.
- **Get Cookie** – searches a file in the database and returns the cookie.
- **Get Size** – searches a file in the database and returns the size.

File Management:

While the file streams are transported across the network, the server is receiving the streams, encrypts them with a password (provided by the client) and saves it in a separate directory.

The directory is consisting of 2 files – the encrypted stored file (named 'F') and the hash file (named 'H').

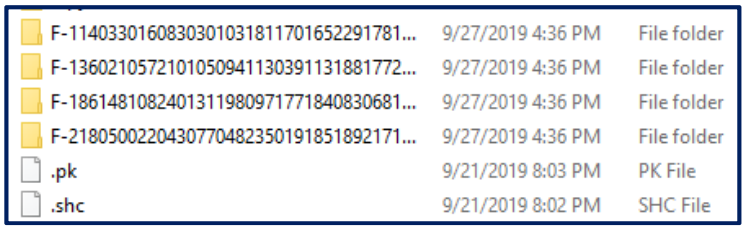
The file is encrypted by the AES encryption algorithm – in which the password, given by the client, is regenerated as a 256-bit key and also a 128-bit vector used in the encryption method.

To get the key we are creating a SHA256 hash from the password, and also to generate the vector we use the MD5 hashing algorithm to get a 128-bit stream, which used as the vector.

In the hash file, we are using the SHA384 hashing algorithm to store the password for user authentication later on.

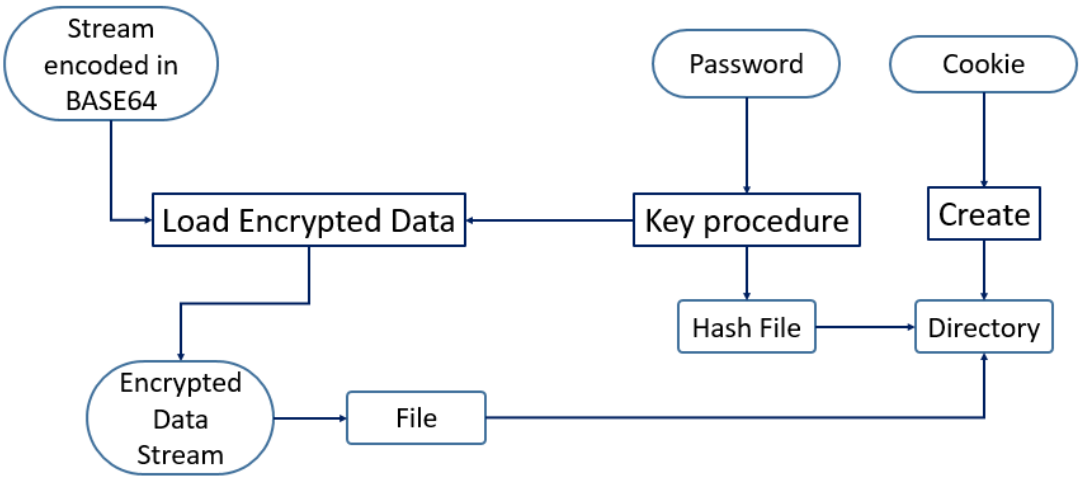
The directory is named as '**F-COOKIE**', in which 'COOKIE' represents the generated cookie (file identifier).

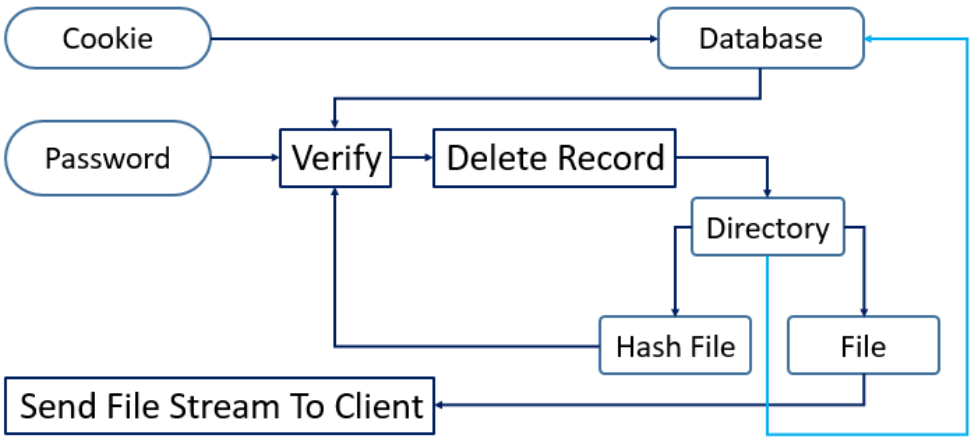
There are 2 additional files to the file system – the RSA key file (named as '.pk') and the hash file (named as '.shc') in which the operation code of the server is stored (via SHA384 hashing algorithm).



The file management unit is making use of these functions:

- **Create** – creates a storage directory.
- **Delete Record** – deletes a record from the database and also removes the storage directory (and the contains files).
- **Key Procedure** – generates the 256-bit encryption key and creates the hash file.
- **Verify** – verifies the user by creating a new hash and comparing it to the stored hash.
- **Load Encrypted Data** – takes an encoded byte stream, decodes it, encrypts the stream with the given key and appends to the file.



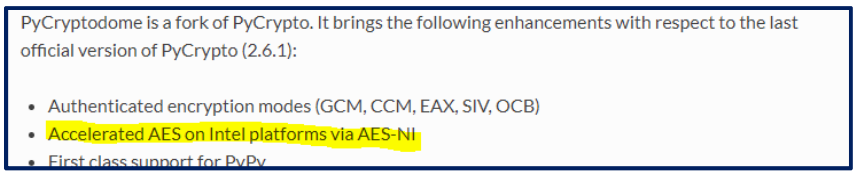


Service Security:

File encryption:

In the file encryption process, the original data is never exposed (meaning the file data is encrypted and loaded automatically, the original data is never stored on the disk).

The library used to encrypt the file is using the AES-NI instruction set (if available) – which used for a more secured and fast AES encryption.



Every data stream is decoded and encrypted individually. Every data stream is in the size of the TCP socket window, and every data stream shall be sent in the original size of the stream to be decrypted successfully by the client.

Socket security (connection security):

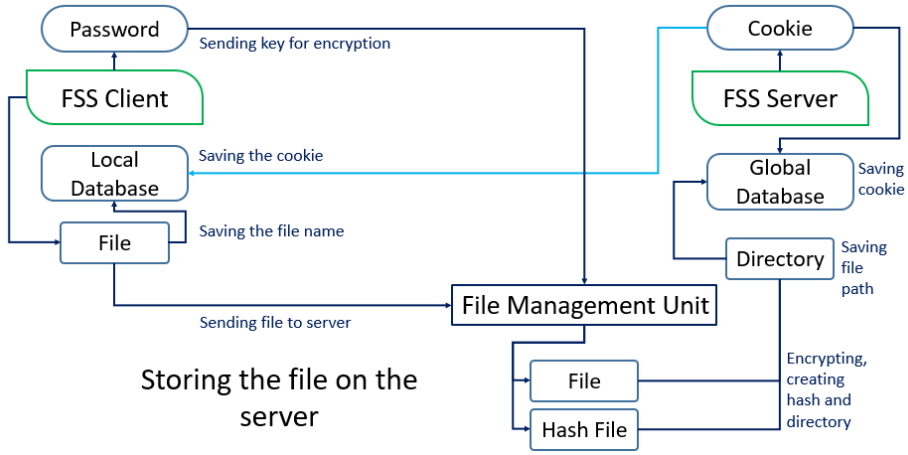
The socket is encrypted via RSA public key encryption… almost.

'PyCryptodome' is using a public key encryption method which is based on the RSA algorithm called '**PKCS1_OAEP**'. but, has a limited data length for encryption (based on the length of the key) which means you need to segmentize the message (if it's bigger then the length limit) and encrypt every chunk of data separately.

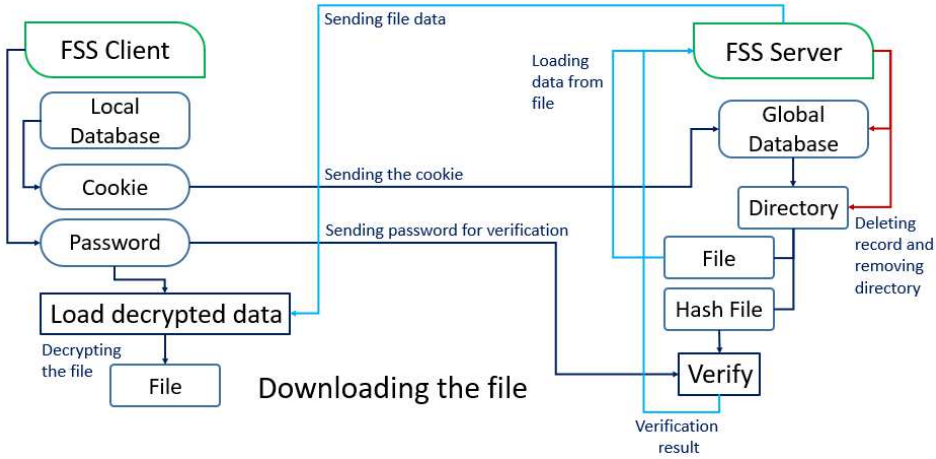
In the first stage of the conversation, the public keys are transported between the server and the client to be used on the encryption.

Because the client and the server uses the RSA encryption throughout their whole conversation, the communication between them is very slow – but, in the future versions the communication will be based on AES encryption (via transporting the encryption keys through an RSA secured socket, and then using the symmetric key for encryption, which is very fast compared to the RSA method). The private key is stored as a file secured by AES encryption.

Over Look:



The client is uploading the file to the server. The client is forming a record on the local database, storing the file path and the generated cookie. He is sending an encryption key, and the server encrypts the data streams one by one. The server is storing the given key on a hash file. The path of the directory is stored in the global database, with the generated cookie.



When the client is requesting to download the file, he is sending the saved cookie, along with the password. The server is verifying the password and if the user was authenticated successfully, the server loads the file streams and sends them to the client. The client is decrypting the file streams and appends them to the file.

Implementations Explained

(FSS Python API)

Before we start…

The python API is using those following libraries:

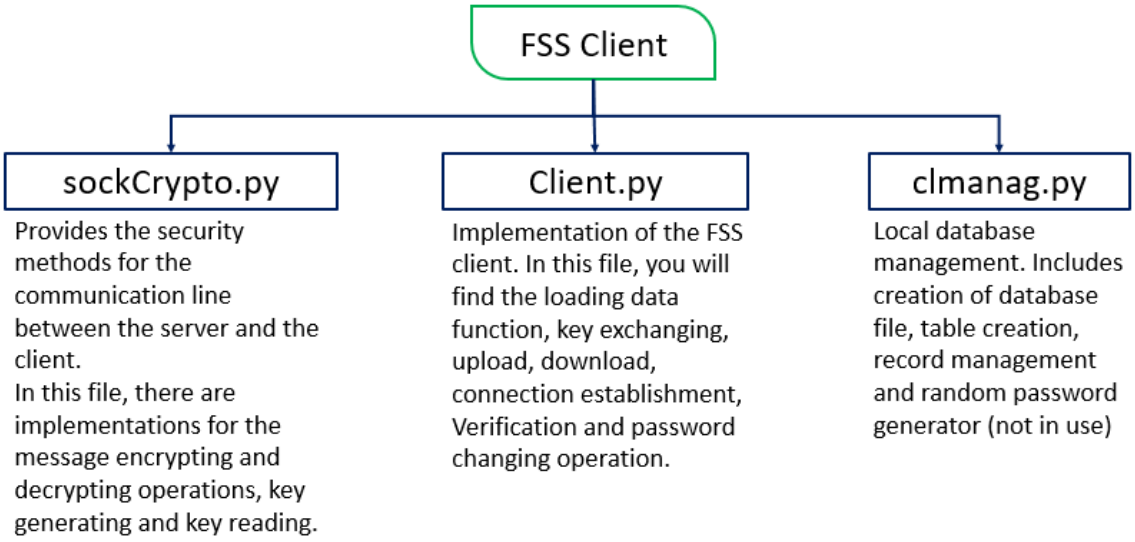
- **hashlib** – used for generating hashes.
- **socket** – used for creating the TCP socket and conversation handling.
- **PyCryptodome** – used for AES and RSA encryptions.
- **ast** – used in the RSA decryption process.
- **os** – used for random number generation (urandom) and file management.
- **traceback** – used for tracking errors (especially for debugging).
- **base64** – used to compares the file data streams.
- **datetime** – used to record the time of saving.
- **sqlite3** – database management.
- **shutil** – used in the record removing process ('rmtree' used to delete the directory).

The FSS API is spread across 7 python files:

- **Server** (server.py)
- **Client** (client.py)
- **Socket security** (sockCrypto.py)
- **Local database management** (clmanag.py)
- **Global database management** (sevmanag.py)
- **Cookie generating** (cookiegen.py)
- **File management** (fmanag.py)

Every file is implementing a part of the FSS service. Every file has a rule, and every file is supplies a certain operation to the server or the client.

FSS Client Implementation Explained:



The sockCrypto.py file is supervised on the connection security. The functions which are included in this file are:

- **gen_key**(password) – generates RSA private key in the size of 2048-bit, writes it to a file end encrypts it with the password.
- **read_key**(password) – reads the private key from the file and returns a two pair of keys: private and public (As RSA key objects).
- **encrypt_message**(message, public) – encrypts a message with a given public key. Returns the encrypted message.
- **decrypt_message**(message, private) – decrypts a message with a given private key. Returns the decrypted message.

The client.py file is supervised on the interaction with the user and the connection with the server. The functions which are included in this file are:

- **exchange**(sock, password) – exchanging the public keys with the server. Returns the public key of the server.
- **load_decrypt_data**(name, stream, password, padding_limit) – loads the encrypted file streams which sent from the server. The function is using the password for the decryption, the padding limit to remove the AES padding and the name (the path of the file) to save the file in the

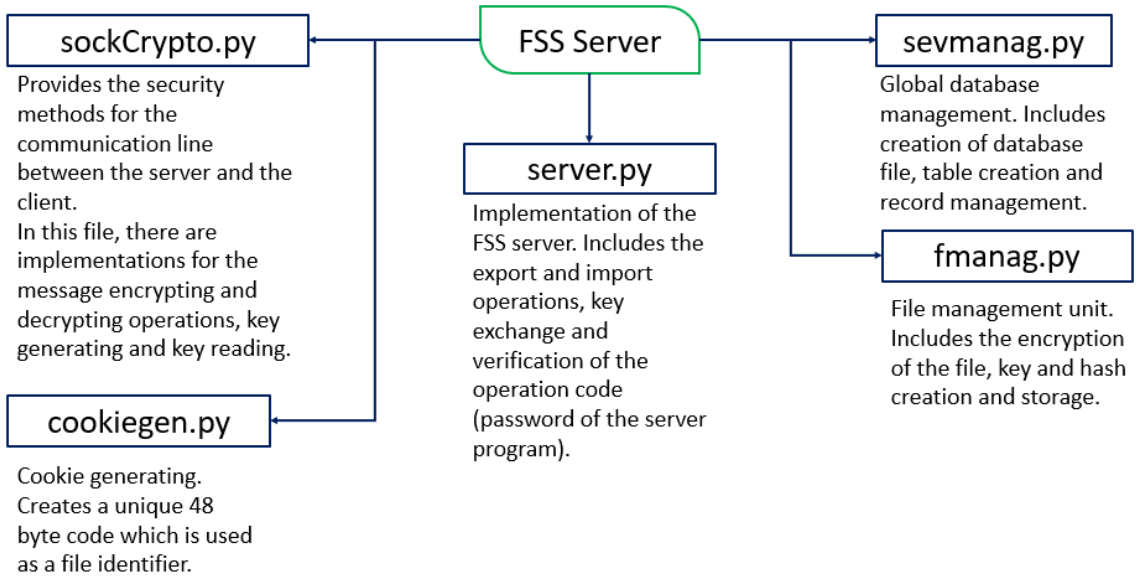
selected location. Updates the padding limit with every iteration.

- **download**(password) – operating the download procedure. In which the requested cookie is sent along with a password, and receives the file back from the server.
- **upload**(password) - operating the download procedure. In which the requested file is uploaded to the server, along with password for encryption. The generated cookie (provided by the server) is inserted to the database.
- **connect**(password) – establishment of connection with the server, along with a key exchanging, returns a socket object and a key pair.
- **client**(password)
- **change_pass**() – changing the clients password and updates the hash file.
- **verify**(password) – verification of the password, returns the result.

The clmanag.py file is supervised on the management of the local database. The functions which are included in this file are:

- **create**(file) – creates a database file, along with a table.
- **exists**(file) – checks if a file exists.
- **insert**(file, path, f_size, cookie) – inserts a new record to database.
- **remove**(file, cookie) – removes a record from database.
- **search**(file, name) – searches a record by the name parameter.
- **list_files**(file) – lists all the records.
- **clean**(file) – cleans the database.
- **get_cookie**(file, name) – returns a cookie of a given file name stored in the database.
- **get_size**(file, name) – returns the original size of a file by a given name.
- **password_gen**() – generates a random string in the length of 32 bytes (not in use).

FSS Server Implementation Explained:



The sockCrypto.py file is supervised on the connection security. The functions which are included in this file are:

- **gen_key**(password) – generates RSA private key in the size of 2048-bit, writes it to a file end encrypts it with the password.
- **read_key**(password) – reads the private key from the file and returns a two pair of keys: private and public (As RSA key objects).
- **encrypt_message**(message, public) – encrypts a message with a given public key. Returns the encrypted message.
- **decrypt_message**(message, private) – decrypts a message with a given private key. Returns the decrypted message.

The sevmanag.py file is supervised on the management of the global database. The functions which are included in this file are:

- **create**(file) – creates a database file, along with a table.
- **exists**(file) – checks if a file exists.
- **insert**(file, path, cookie) – inserts a new record to database.
- **remove**(file, cookie) – removes a record from database.
- **path**(file, cookie) – searches a record by cookie, and returns the path of

the storage directory.
- **clean**(file) – cleans the database.

The fmanag.py file is supervised on the management of the server file system. The functions which are included in this file are:

- **create**(cookie) – creates the storage directory.
- **delete_record**(db_file, cookie) – deletes the file record and the storage directory.
- **key_procedure**(directory, password, create) – creates a 256-bit key and a SHA384 hash to store in the directory.
- **verify**(directory, password) – verifies a given password with the hash stored in the selected directory.
- **load_encrypted_data**(directory, stream, password, create_hash) – loads a BASE64 encoded file streams, encrypts these streams and saves them in the storage directory.

The cookiegen.py file is supervised on the cookie generating process. The functions which are included in this file are:

- **gen**() – generates a 48-byte sized strings (made of decimal digits).
- **check**(file, cookie) – checks if a cookie is already used.
- **get_unique_cookie**(file) – returns a unique cookie code.

The server.py file is supervised on the operating of the server and interaction with the client. The functions which are included in this file are:

- **server**(sock, password) – operates the server.
- **exchange**(sock, password) – exchanges public encryption keys.
- **operate**(sock, request, private, public) – operates a function according to client's request.
- **importing**(sock, request, private, public) – importing file.
- **exporting**(sock, request, private, public) – exporting file.
- **verify**(password) – verifies operation code (server's password).
- **change_code**(new) – change operation code (not in use).

To Close Things Up

Now, as you can see, this documentation describes a pretty simple protocol which can be implemented in many ways and also can be improved.

There are things which I, unfortunately, wouldn't have the time to change or improve. The problems with this protocol are the long conversation time, caused by the RSA encryption, and can be drastically decreased if we just used the AES encryption, along with a key exchange above the RSA encryption. Another problem is the cookie code, which I preferred to create a randomly generated string but, it would slow up things. I offer to generate the cookie as an index number, meaning that the identifier will be the index number of the created record on the table. The next thing is to create a built-in file compression, to make the loading much faster and also let the client load not just one file at the time, but an entire directory. Also, a multi-threading to manage several connections simultaneously needs to be added to the server to make the service fully operational.

Another thing is to implement a user-based service, which is not hard to implement and maybe will show up sometime in future versions of this protocol.

Special thanks to Rony Alaluf and Tamir Grossman, who helped me a lot in this 1-Month project.

I will be glad to see where this project goes and develop,

Thank you for all your time,

Orel Aharonov.

Contact:

You can contact me via GitHub: https://github.com/odev954

To access the project files, you can find them on GitHub: https://github.com/odev954/the_FSS_project