File Transfer Expanded

Introduction

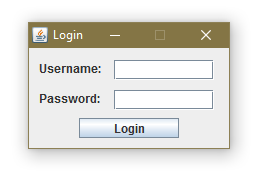
This project is a raw cloud file-server implementation. The user is able to store all kinds of files and data. The main feature of the project is that the user can store **very large files (>200 GB)** without bothering for the **integrity** of it due to **continuous hash checking** while sending the file to the server. The encryption is the second large feature that is available. This encryption uses the standard **RSA public / private key** for authentication, then the **AES channel** is started after communicating the key.

User Key Features

* Upload large files.

**End to End encryption**

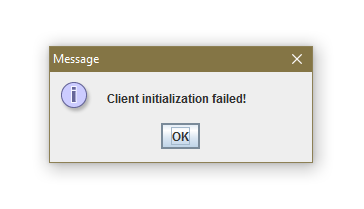
* Download files.
* Remove files.

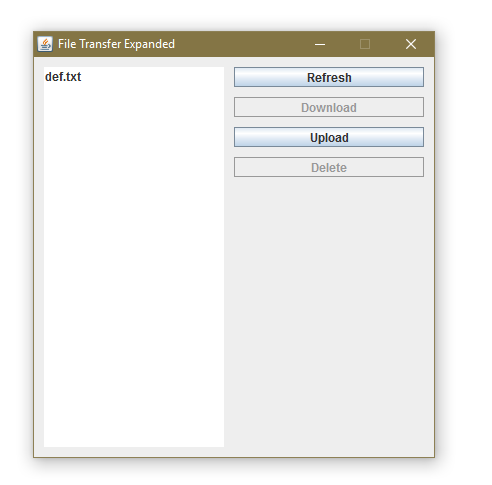


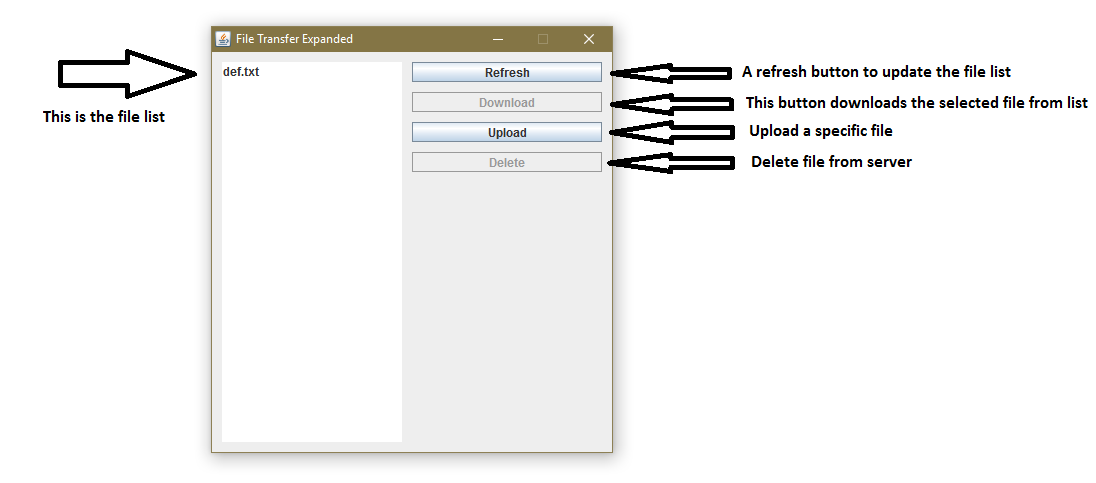
The Application

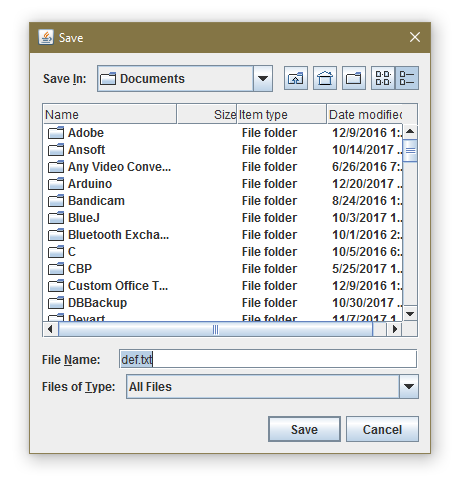
The application loads a text file “connection.txt” which contains data about the server: Ip address and Port. If the file does not exist, a default one is created with the address “127.0.0.1” and port “688”.

Next, the login form spawns and asks the user for credentials.

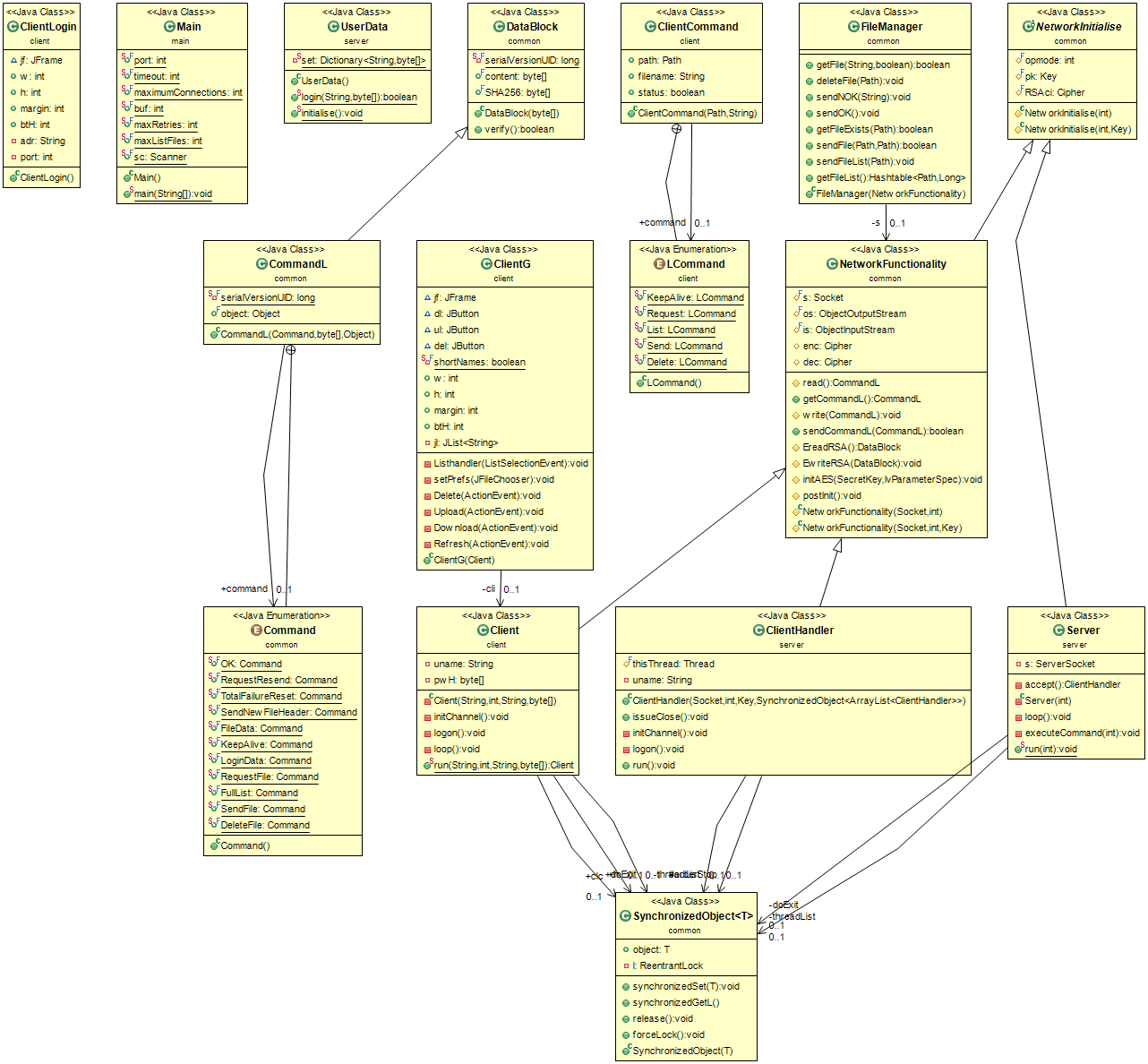
If the connection failed, credentials are not correct or encryption process failed, an appropriate message will spawn on screen before the program quits. Details are printed to the console and at least one exception will be thrown. Such messages can be:

After the user successfully logged in, the main window will appear:

The GUI is user friendly and simple.

For the **Download** and **Upload** buttons, an additional file browser is to be used in order to select the download location or the file that needs to be uploaded.

**Some limitations** are not to be ignored. One of the drawbacks of this application is that it is currently impossible to create folders on the server. The only folders present on the server are the user folders. Each user has its own folder. These folders may/must be space-limited by the operating-system in order to prevent running out of disk space.

Classes and Functionality

**Basically the server spawns threads for each client connected.**

This is ok because if one thread crashes, the server will be still up and running and it improves speed on multicore servers.

A drawback is the memory, as each thread sends files or gets files concurrently, more memory is needed per thread. A quick fix would be limiting the maximum concurrent send/get runs.

**After spawning the thread, the encryption channel is to be set up.**

Encryption is of maximum end-to-end security, first step is the RSA, to communicate the randomly generated key for the next, AES encryption. **All Data/Login/Channel Initialization is integrity verified and packets are resent if verification fails.**

There may be small speed decreases because of the need to encrypt/decrypt/verify data.

**The login occurs on the encrypted channel.**

User credentials are Username and Password Hash. Hashing the password is just a security protection, for both the communication channel and server database storage.

If logon/encryption fails, the user is automatically disconnected and it is required to restart the application.

…

Critical Vulnerabilities

The encryption method uses SealedObject class that is prone to **RCE and Overflow** if used on a network channel, problem of ***deserializing a malformed object***. A fix is to use byte array chunks and then provide manual serialization/deserialization.

File checking for sending and receiving files is partially implemented. That is, the server can receive name buffers too large and probably crash the current connection.

…

Future Developments

Ordered by severity, fixing the deserialization problem, debugging very large file names, finding a memory management solution for the large amount of threads, not letting the same user connect several times at the same time, implementing the ability to create folders and group files, displaying status about file transfer (remaining time, percent transferred) and implementing the app for mobile usage (Dropbox like) is almost the entire list of future developments and bug fixings.