ENIGMA

A Python-based file sharing server that ensures privacy, anonymity and confidentiality

Xavier Torrent Gorjón

Resum—Des dels inicis de la humanitat, la comunicació i la seva privacitat ha estat un camp d'exhaustiva recerca. Això encara és cert avui dia, considerant que la privacitat i seguretat a Internet és motiu de debat diàriament. Aquest projecte apunta a oferir un servidor basat en Python capaç d'emmagatzemar arxius en línia, tot preservant la seva privacitat, confidencialitat i anonimat. Hem iterat sobre diversos dissenys en la nostra cerca d'una implementació que complís tots els nostres requeriments. Finalment hem aconseguit una solució que implica usar dos servidors diferents, un a càrrec de les accions com logins i processos de registre, i l'altre responsable de tractar amb els fitxers. Des de l'esquema teòric hem desenvolupat una aplicació considerant tots els possibles atacs que es podrien fer contra el nostre sistema, centrant-nos en els atacs de man-in-the-middle. Hem completat satisfactòriament el projecte en el calendari previst, oferint un producte que pot complir tots els requeriments de seguretat d'avui en dia, mantenint la senzillesa d'ús.

Paraules clau—Privacitat de dades, confidencialitat, anonimat en arxius.

Abstract—Ever since the inception of humanity, communication and its privacy has been a field of extensive research. Even today, this premise holds true, as privacy and security on the Internet is a daily topic. This project aims to deliver a Python-based on-line server capable of storing all kinds of files while preserving privacy, confidentiality and anonymity. We have iterated over many designs in our search for an implementation that met all of our security and usability requirements, coming up with a solution that involves the usage of two distinct servers, the first one in charge of the user-related data, such as register and log-in tasks, and a second one responsible of handling the actual files. From the initial theoretical scheme we have developed an application considering all the possible attacks that could be made against our system, mostly focused, but not limited to, man-in-the-middle attacks. We have successfully completed our project in the scheduled time, delivering a product that can stand up to today's applications security requirements, while still offering a great ease of use.

Index Terms—Data privacy, confidentiality, file anonymity.

1 Introduction

From the Caesar cipher in the ancient Rome to the Axis' Enigma machine in WW2, cryptography has been a subject of vital importance on human history. Being able to transmit information readable only by a closed group of parties is the basis of hundreds of today's computer applications, ranging from instant-chat programs to bank accounting systems. Taking this one step further, we sometimes also want our information to be anonymous, that is, that the author of a particular file can't be traced back.

Anonymity can be a powerful weapon on the Information Age we currently live in. It allows people to express their inner thoughts without fearing a retaliation from an opposite force, be it a government or a competitor. While anonymity has its drawbacks as it can be used to cover illegal actions, it should be a right to every single individual to be able to share their thoughts or their possessions freely using the Internet as the ultimate communication platform.

Starting from this belief, we present this project that aims to fulfill all the security requirements of a modern application to allow a certain individual, group or community to share their thoughts and files freely on the Internet, resting assured their authorship remains secret and their data are only available to the ones they choose.

This project will be primarily focused on obtaining a robust solution to the presented problem, guaranteeing privacy and confidentiality over anything else. When that requirement is fulfilled, a secondary goal will be to make the application as friendly as possible, from the user's

- *Contact E-mail: xavier.torrent@e-campus.uab.cat*
- *Specialization branch: Tecnologies de la Informació.*
- *Project mentor: MªCarmen de Toro (dEIC)*
- *Year 2013/14*

point of view and also from the sight of a programmer willing to improve or maintain the code.

This article will start with a brief description of the current State of the Art concerning our project. We will then describe the preliminary considerations taken, as well as establishing the objectives that must be fulfilled in order to evaluate this project's success. Then we will move on onto more technical details regarding the actual implementation of our solution, explaining every step taken on the design of the communications and data structures. Finally we will conclude this document with a conclusion section, evaluating which requirements and objectives we have successfully achieved.

2 State of the art

Currently there are many file-sharing services on the Internet that claim to provide satisfactory levels of privacy or anonymity.

One example of one of these companies could be Mega. While they have recently released an SDK to develop applications using their services, the actual implementation of the server itself its not of public domain. Whether it fulfills our requirements of not only privacy and confidentiality, but also anonymity, is not clear. Particularly, they do not make any claim to offer an anonymous service.

Another on-line service related to our project could be Anonfiles. This website offers the possibility of uploading files anonymously, but does not offer any kind of encryption or personal account services.

Our application seeks to be an outstanding service among the currently provided on the Internet, by ensuring security and anonymity at all stages of each file's transfer and storage.

3 Preliminary Considerations

Before we start explaining the actual project, we must first clarify some aspects of our work such as the current cryptography regulation on the EU or why we have chosen Python as our development language, as those points have a high impact on our application and strongly define our course of action.

3.1 European Union cryptography regulation

As strange as it might seem, cryptography technology receives a similar treatment as army equipment such as tanks or stealth-fighters. Our first goal would be to be able to export this software inside the EU. If this proved feasible, a subsequent expansion to countries outside Europe such as the USA, Russia or Japan would be of a great interest.

Cryptography regulation can vary from one country to another, even inside the EU. However, there is an international treaty called Wassenaar Arrangement [1], signed by 41 different countries, that seeks to provide a unified regulation to cryptography, as well as other technologies. Cryptography regulation is defined on Category 5—Part 2 of the treaty, "*Information Security*".

Until December 2000, symmetric cryptography software was restrained by a limitation of its key size to a maximum of 64 bits [2]. Starting from this year, this limitation has been taken away, allowing the export of mass-market software of any key size.

3.2 Choosing Python as the development language

One of the initial decisions to make was to select an adequate programming language that could fulfill our expectations regarding performance and ease of use.

A first concern was to choose a language capable of working under an object-oriented paradigm. From this initial condition we pre-selected C++, Java and Python as our languages of choice to develop our application.

In an initial research, cryptographic functions for data encryption showed a much higher performance on a compiled language such as C++ rather than those written on Java or Python [Fig. 1].

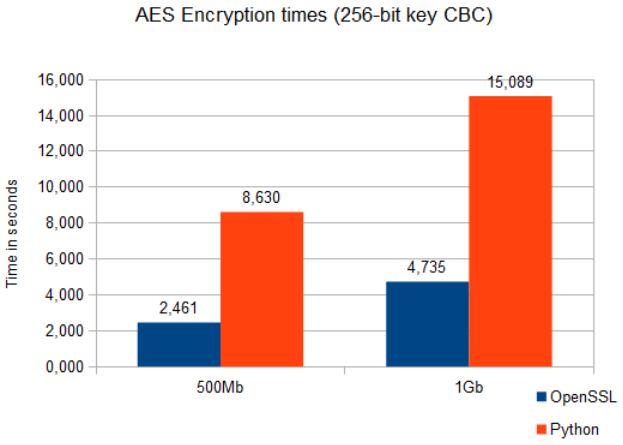

Fig 1. Pure Python vs OpenSSL encryption speed comparison. OpenSSL shows 2-3x faster encryption.

As a drawback, C++ implementation proved to be a much harder task as the application deals with a lot of different data structures and communication exchange that was much easier to program and maintain using a higher-level programming language.

All of this lead to our actual design, that involves using Python for the most part of the application, including the TCP communications and data structures, while using an external library, OpenSSL, to cope with the encryption burden. This allows us to benefit from the best of both options, writing an application easy to modify and update while using an external library written in pure C for the heavy-computing tasks.

Regarding our choice on high-level programming languages between Python and Java, our decision answers to our belief that Python code is easier to read, which means it will be easier to re-write or improve if needed.

3.3 Cryptographic algorithms selection

As with the programming language selection, we have a wide variety of cryptographic algorithms to select from as we develop our application, for both symmetric and

asymmetric encryption.
Our usage of asymmetric algorithms will be initially limited to the first steps of clients' connections to ensure server authentication and to provide a basis for sharing each connection symmetric encryption key between both parties. For this purpose we have selected the RSA algorithm, which provides both encryption and authentication functionality.
In regards of the symmetric encryption algorithm that we will use to cipher our files and communications, we will stick to the AES standard. By using a key size of 256 bits we will ensure that our application remains usable in the foreseeable future.

4 OBJECTIVES

In this section we will be discussing the objectives of this project, from the vital objectives required for the application to work, to the secondary goals like the whole design and implementation of a friendly user interface. We will also provide a table indicating their overall importance to the project [Tab. 1].

4.1 Security against man-in-the-middle attacks by encrypting all communications

One of the main concerns of today's applications is to deal with traffic sniffers and malicious third-party attacks. The security in the application communications' is of a critical importance. For this reason we will be encrypting not only the files users' upload, but also all the packets' payloads, to ensure no attacker can gain any kind of information that could compromise the systems' security or anonymity.

4.2 Storage privacy by using local encryption

All files must be encrypted and decrypted on the client's system, as it is of a vital importance that not even the file server has knowledge of what is being uploaded or downloaded to preserve privacy.

4.3 Storage anonymity by separating user data and file data

In order to ensure storage anonymity, we must separate the data relative to user's account from the actual files uploaded to the server. We propose a model where this separation is made by storing these components in different servers. This way, even under the condition of a man-in-the-middle sniffing attack or a server security breach, no assumption can be made about who uploaded a certain file.

4.4 User interface design and development

Whilst the core functions of the application can be easily used from a text-only environment, it can come handy to inexperienced users to have a graphical user interface, as they often are more self-explanatory. With this purpose in mind, we aim to provide both environments, so that everyone can freely pick the one that suits them better.

4.5 File sharing: import and export

Since one of the principal motivations for this project is to provide users a platform that they can use to share their files, the implementation of a file-sharing protocol besides the actual upload and download actions is in order. However, this highly depends on how we want to focus the application: the sharing protocol might vary greatly depending on if we are building a mass-market sharing application, or if we decide to develop a program focused on one-to-one file sharing.

4.6 Keeping the application platform-independent

We believe everyone should be able to use our application, regardless of the operating system being used. A decision was to be made between using a regular desktop-based application or a web-based application, since the time designed for the project was short to do both.

A desktop application developed using free software allows the end user to change, at the very least, the appearance of the program's interface, while a web-based solution needs to be downloaded every time from a designed website, which limits what users can customize about it. Seeing this advantage from the desktop-based solution, we decided to chose it over the web-based application. Nevertheless, having chosen a desktop-based solution using Python does not limit our application to be further improved with a web client.

TABLE I. OBJECTIVE RELEVANCE

Objective	Objective Relevance	
	Vital	*Secondary*
A) Security against MITM attacks	X	
B) File storage privacy	X	
C) File storage anonymity	X	
D) User interface development		X
E) File sharing		X
F) Platform-independence		X

5 METHODOLOGY

The methodology used for the development of this project will be centered around the divide-and-conquer strategy. The application will be divided into many distinct components -whom some of them might be reusable for other projects- with each one having a clear purpose.
Focusing on implementing a modular solution we can guarantee that our code will be easy to maintain and be further developed on a hypothetical post-project work. Having a modular implementation is specially important in our servers, as we will be dealing with a lot of different threads that can extremely increase the difficulty of developing and testing our software if not programmed carefully.
In this regard, the project will be divided into five modules, defined in the following subsections.

5.1 Cryptography module

The first module of our development schedule will be the one containing the files required for cryptography purposes. This module will be a basic adapter between Python code and OpenSSL console function calls. Since both client and servers will require cryptography usage, this module will be present on both sides, although servers won't use the functions designed for file encryption purposes.

5.2 Communication handler module

To allow communication between the clients and the servers we will need a module to handle the connections between them. We will be using TCP sockets for this purpose, as we need a reliable data transfer protocol for our application. This module will be directly related to the cryptography module, as most of its functions will require cryptographic utilities to function.
Servers will use a fairly different connection handler module, but some of their functionality can be reused.
Coming in the following sections, we will also explain the development of the protocol we have designed for the client-server communication.

5.3 Data handling module

The application needs various data structures on both client and servers, and this module will aim to unite them in an ordered way. Data required by the application will range from the login variables to the actual files stored on the file server.
These modules will be completely different on both sides, and there is little chance anything can be reused.

5.4 Main application module

A module that groups up both communication handler and data handler modules is needed to enable our data to flow between modules. The main application module will be a bridge between both components, and will offer the basic text-mode user interface, from which we will later build our graphic interface.

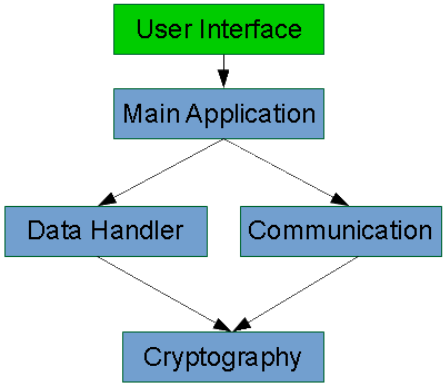

Fig 2. Module dependency overview. Both the client and the server will use a similar structure.

5.5 Graphic interface module

Upon completion of our main application module, we will need another additional module to provide a Graphical User Interface (GUI). This will be done using a Python library called TkInter. While there are many other libraries for this purpose, TkInter comes installed by default on many Linux distributions and is also easy to install on both Windows and Mac. The application won't initially support a GUI for the server side, due schedule restrictions.

6 Development Planning

In order to facilitate the software development and avoid any possible security breaches, an extended design time was required. The project was intended to be developed over the course of 13 weeks from its start on October 2013.
Over the course of the first weeks the main focus would be to develop and test the module concerning encryption, as well as starting a research to list the possible legal restrains that a cryptographic application might have under export laws.
Starting on week two, the design of the other modules from the application would began. After the design was complete the implementation should start. For these modules an extensive time was assigned, summing up to 9 weeks.
Last two weeks of the designed time were reserved for writing this documentation and performing various tests over the written software.
On Figure 3 there is a Gantt diagram representing this project's planning.

7 Communication Protocol Design I

Following the module design, a communication protocol design is in order. There are many factors on this section that we need to address.
First of all, we need to picture the communication requirements of our application. As we briefly explained on the introduction, the system will be composed of two different servers performing different tasks. From now on, these servers will be called Data Server and File Server.
Data Server will be in charge of storing all the data concerning users. This includes user names and hashed passwords, the encrypted files holding the information of each user's files and temporary information derived from each user's connections. Data Server will also need to be capable to communicate with the File Server on demand, since all file uploads and downloads will require a user validation.
File Server will store the users' encrypted files. Every time a user attempts to upload or download a file, the File Server will check for a previous verification from the Data Server concerning that action.
In order to ensure anonymity, we have to minimize the data sharing between both servers. When a user wants to download or upload a file he will first connect to the Data

Server to verify his credentials. If validated, the Data Server will send a simple token to the File Server stating that a user will soon make a request for a certain file, which must be accepted, without stating an exact user or IP direction. To ensure no possible race-condition attack here is made, the same token will be given to the user as well, allowing him to prove to the File Server he made the legit petition.
The application will initially support seven different actions: connection, register, log in, log out, file upload, file download and a sharing feature.

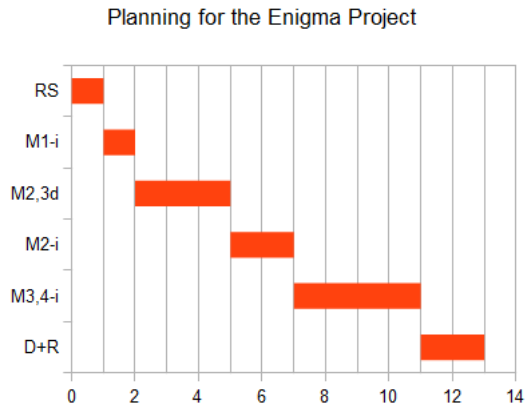

Fig 3. Planning for the Enigma Project. Titles read as follows: RS: initial research, M1-i: Cryptographic module impl., M2,3d: Communication and Main modules design, M2-i: Communication module impl., M3,4-i: Main module and GUI implementation, D+R: Documentation and Review.

7.1 Connection action

The connection action will provide a basic key exchange using the known server's public key to establish a shared AES key that will be used from this point onwards. This will be similar for both servers' connections.

7.2 Register action

The first step a user will need to make to use the system will be to register himself in the platform. As strange as it may seem for an application that tries to provide anonymity, it is required in order to save a list of the user's files on-line and be able to recover it later. The register action will ask for a user name, a password and a valid e-mail address. This e-mail address will no longer be used unless the account needed to be recovered (e.g. in case of loosing the password), so it is possible to use a temporary e-mail address.
This step will require to be validated through a confirmation code sent to the user's email (encrypted with a user's public key generated on this step, to prevent impersonation attacks). This code needs to be submitted on the application to complete the registration.

7.3 Log In action

Once register step is completed, users will be able to perform a regular log in into the platform using the Data Server to verify their identity and download the list of files they currently have access to.

7.4 Log Out action

Once a user has finished all the needed file transactions, he will have the option to log out from the platform. This action will also be automatically performed in case of closing the application without using the log out feature, to ensure no dead threads are left on the Data Server (other protections against this will be implemented server-wise as well).

7.5 File upload

After verifying the user is correctly logged in upon an upload request, the Data Server will generate a file ID for the current petition, sending it as a token to the File Server. When the File Server sends a confirmation message back, the Data Server will send this token to the Client, which then will directly connect to the File Server to upload the file. Once the upload is completed, the application will automatically add the file to the user's list of files, also updating Data Server user's information.

7.6 File download

In a similar fashion to the file upload action, downloading a file will initially use the Data Server connection to ask for a valid token to send a petition to the server. This action will not require to update any sort of information on the user's file list.
A visual example of this process is shown on Figure 4.

7.7 Sharing feature

While there are many possibilities when designing a sharing feature, we decided to go for one of the simplest: obtaining a string that identifies a certain file with its ID and password. The reason behind this decision is that this is the easiest way for everyone to share their files on the Internet, using a similar feature to those used by a lot of file-sharing websites.

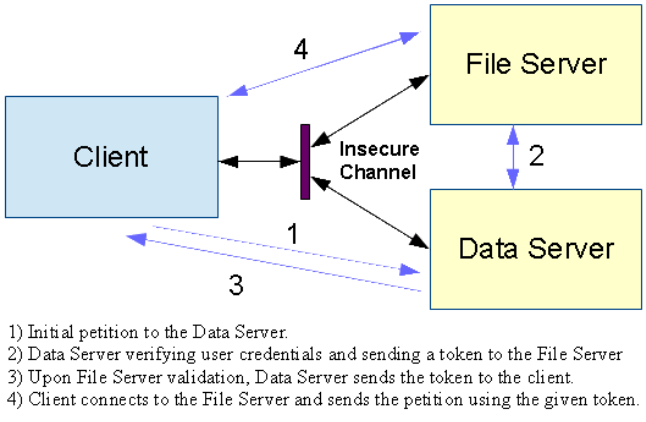

Fig 4: Download command example. Blue arrows represent the different connections being used, while numbers represent in which order they are performed.

8 Communication Protocol Design II

In this section we will provide a brief overview of the message format used by the program on the application layer of the TCP/IP model.
Each message will start with a header made of two constant-sized values: the OpCode field and the Length field. OpCode will provide information of the actual action being processed, while Length field will contain information on how large the payload is. OpCode and Length fields will have a length of 4 and 6 bytes, respectively.
After each header there will be a payload string assuming the Length field is different from 0. This payload will be always encrypted regardless of the message type, even if the data to be sent is already encrypted, to prevent attackers from getting information such as *which encrypted file* someone has uploaded or downloaded. When payload is composed by different variables (such as on a register action, where a payload will be composed by a user name, a hashed password and an e-mail address), the data will be packed using Python's cPickle serialization module.
Following on Table 2 some message examples are given.

TABLE II. Communication Protocol Overview

Client – Data Server Messages			
Message Type	*OpCode*	*Length*	*Payload*
Key Exchange	0000	size(payload)	key
Register Step 1	1000	size(payload)	cPickle{name, pass, mail}
Load File List	6000	0	N/A
Data Server – File Server Messages			
Message Type	*OpCode*	*Length*	*Payload*
Upload	3000	size(payload)	fileID

The whole application will be developed trying to make it header-length independent, so that if future changes require larger headers, the application does not need to be re-programmed.

9 Application Design: Client

All possible client actions have been defined in Communication Protocol Design I. From this starting point we will develop a program that will offer a simple command-line interface to perform the communication between the user and the system. This interface will be based on a simple iteration, parsing the strings introduced by the user one at a time and executing orders accordingly. Client's application main thread will perform this task, while creating and executing additional threads [Fig. 5] to process some of the commands, as this will be a requirement in order to avoid the feeling that our application gets frozen.
Client application will need a moderate amount of data structures to perform tasks such as log in or file list handling. On the latest release, the client features the following data structures:

- Log-in information: A client-side verification of the current log-in status. This can be hacked on the client side, but its just a feature for legitimate users, as Data Server will have a duplicate of this information. This includes the AES key used for the communication with the Data Server, and each subsequent AES key generated for the file transfers.
- List of owned files: Each owned file requires a tuple made of its name, its key for the decryption step and its file identifier on the File Server.
- GUI variables: The graphical interface needs an additional set of variables to control the data flow and to enable/disable some buttons upon taking certain actions (e.g. not letting a user download files before a log-in is performed).

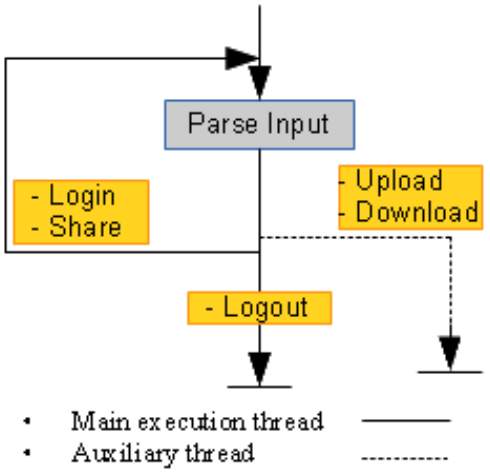

Fig 5. Client execution flowchart after connection.

10 Application Design: Servers

10.1 Data Server

Data Server will feature a single TCP socket that will receive connection attempts from client applications. Once a new connection is established, it will be handled on a secondary thread that will manage all the upcoming commands from that particular user.
Every connection will start with an AES key exchange between both parties, using a public key cryptography scheme using the server public key. This key is meant to be published on a secure website, so everyone that attempts to use our application can verify the public key used on the client is the legitimate one. This security feature is designed to block impersonation attacks on maliciously edited Enigma clients.
Child threads will be able to directly access the user database, which implies they will not need additional modules to perform the tasks of registering, log in and updating the file list of a certain user. File uploads and downloads, however, will require additional functions to communicate with the File Server to send the transfer token.
These threads will also be able to detect broken pipes,

ending themselves in case of a client disconnecting abruptly.
Each thread server will feature the following data structures:

- Log-in information: The same structure described on the client application.
- Lock list: These Python locks will be used to prevent race condition issues on the server application (this is further explained in Section XII-E).

10.2 File Server

File Server will actually be composed of two listening sockets. The first socket is the one covering client's incoming connections, while the second will be a private socket used only to communicate with the Data Server.
When the File Server receives a token from the Data Server through the second socket, it will assign it to the file name that's been asked to be uploaded or downloaded. Once a connection from a client attempts to download this file, the File Server will grant them permission and the transfer will begin. Any client connection attempt asking for a file that has no token assigned to it will be instantly rejected.
Figure 6 shows a sequence diagram explaining the communication process between both servers when there is a request from a user to upload or download a file.
Since File Server only works with file transfers and all the processing is done on the Data Server, it needs no specific data structure to perform its actions. This is a proof that the user data is being completely separated from the files.

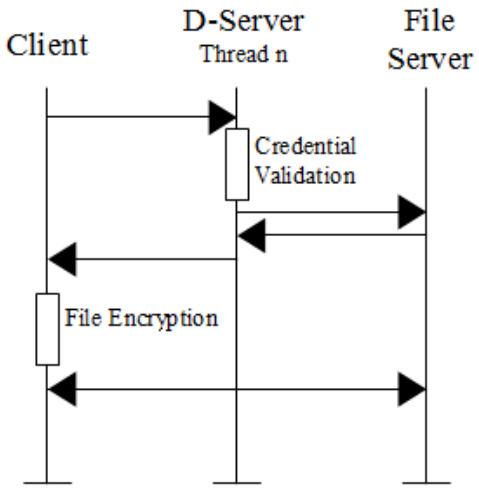

Fig 6. File upload/download communication sequence diagram.

11 Application Design: Graphical User Interface

From the very beginning our goal was to create an interface that was simple enough to be used by any kind of user. For this purpose we minimized the number of windows needed to work with our application, as shown on Figure 7.

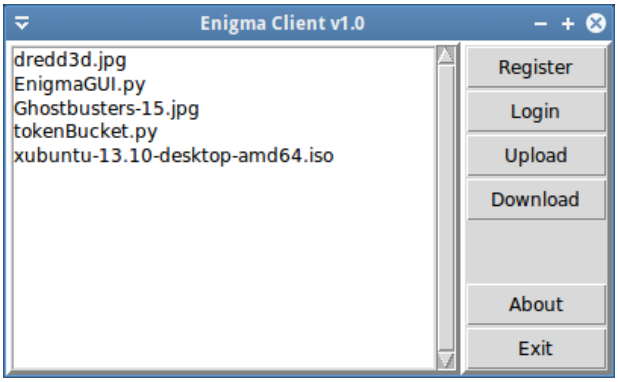

Fig 7. The first iteration on the Enigma Client GUI.

The main window displays in a column of buttons located on the right side an interface to all the functions of the main application module. On the left, a list of our files is shown.
When clicking on the Register or the Log-in buttons an additional window appears to show the input fields required for those commands.
In a similar way, Upload and Download buttons open a file selector that comes as an additional feature of the TkInter library, saving us the burden of manually programming it. This file selector lets us select a file -or a destination, if downloading a file- anywhere in our file system, allowing the user to decide where he wants to store each file.
All the commands on the user interface execute their actions on separate threads if their processing time is longer than a few moments. This allows the user to upload or download many files simultaneously, while preventing the interface from freezing completely until the command is finished.

12 Major Drawbacks During Development

The large amount of time dedicated to the design of the application proved to be invaluable when the time for the actual development came. Although the main design was never changed, some changes had to be made on the road due specific restrains of Python as a programming language or other factors. In this section we will discuss those who were harder to deal with and the workarounds made to solve or mitigate them.

12.1 String encoding on Python networking

The OpenSSL library produced some random binary strings that sometimes couldn't be sent through the Python TCP socket implementation, and returned an error regarding failures reading the string. To solve this problem we deployed a pure Python implementation for the random number generator, intended initially as a temporary solution. However, it proved to function perfectly, so it lived through the different versions until the release.

12.2 Python Global Interpreter lock

Python threads cannot be executed by various processors. This is an ongoing limitation that seems to have no estimated fixing time by Python developers. This handicap is caused by the Global Interpreter Lock, which intentionally prevents the CPython interpreter from executing the same bytecode on different processors, as CPython's memory management is not thread-safe [3].
While this issue does not restrict our clients performance -as we encrypt our files with calls to the OpenSSL library, which is the most computing-intensive task on the client application-, it might become a problem for our servers if the user population grows to a point where memory or computing limitations start to show up. To overcome this limitation we could use Python's Multiprocessing library, changing our threads to independent processes. Nevertheless, this would require a redesign of the server, since an independent process wouldn't be able to access to the main thread's variables.

12.3 Linux TCP buffer size limit

As our servers are intended to be ran on a GNU/Linux environment, we need to make sure the system is ready for a large number of heavy traffic connections. Most common GNU/Linux distributions come with a 128KB TCP buffer size. While this can be enough for a desktop user, on our server's case might be not enough.
Increasing this limit is as easy as editing the *wmem_max* and *rmem_max* values on */etc/sysctl.conf* using the GNU/Linux console [4].

12.4 Dealing with unexpected disconnects

On the firsts releases, the Enigma servers suffered from keeping useless threads alive when some connection broke or some command went wrong (e.g. when a user disconnected by closing the application without using the Exit command). This was a severe problem due the fact that these threads would never naturally die until the application was closed, consuming system resources that went into waste.
To deal with this issue, the receiving sockets now are always checking for zero-length inputs, which are an indicator that the connection is actually broken. With a simple *if* statement we can solve the whole problem, causing the thread to end when that 'null' income is detected.

12.5 Internal race conditions

Considering many threads are running simultaneously, some of them might want to access the same data in a brief span of time. This fact could lead the application into a *race condition*, causing unintended behavior or even data corruption.

Python threading library offers a solution to this in the form of Locks. Whenever a certain function is vulnerable to this issue, we can block the execution of this code by other threads by acquiring a certain Lock. This blockage will be active until the same thread who blocked it releases the Lock, enabling other threads to take control over it.

13 Project Conclusions

Overall, the project has been completed satisfactorily on the scheduled time. We have successfully developed an application capable of performing the designed tasks accomplishing our requirements.
We were surprised to notice how a well-performed and extensive design stage could lead to having way less trouble when developing the application. The large time designated to planning the application really payed off when the moment to write it down in Python code came.
Another positive conclusion from doing this project has been observing the vast amounts of Python information and documentation available on the Internet. Every time the development got stuck by something, it was extremely easy to find out information on multiple websites about how to solve the problem.
However, above everything else, this project can be used as proof that it is possible to deliver high-quality applications using free software. Using GNU/Linux as our development environment of choice, every library and program used to produce this application and its documentation is released under free software licenses such as GPL.
Choosing Python as our development language has been extraordinarily useful towards achieving our goal to make the application platform-independent. The latest versions of the application can work out-of-the-box on most Linux distributions and Mac computers, and the only requirement to make them work on Windows is to download a Python interpreter and any library which could be not included in the interpreter. A Java implementation might be better in this regard, as most Windows systems have the Java Virtual Machine installed due Java being a more widely used language for Windows applications.
From a practical point of view, the application looks very easy to use, and the mouse commands feel responsive. If another interface was to be made, a good course of action would be to make a poll, asking different kinds of probable users to explain what they feel the actual interface lacks. The fact that the interface design is completely separated from the functions implementation lets experienced users and programmers to freely redesign the interface without needing a lot of background on how the rest of the application is programmed. Another benefit from this is that it would be relatively easy to build an interface in any other programming language and bind it to the internal Python function calls.

14 Future Work

Considering the last version of the application, there is still a lot of room for future improvement. In this section we will discuss some of these features that didn't make it to the final version due schedule limitations.

14.1 Allowing upload/download speed limitation

Possibly one of the most useful features not yet implemented is the transfer speed limitation. This option comes handy when sharing a network with other users, or when

performing other network-dependent tasks -like on-line calls- while using the Enigma Client.
The standard TCP implementation is not concerned about the actual bandwidth being used. This may often cause experiencing lags on applications where latency matters. There are, at least, two ways to solve this problem.
The first solution involves tricking the TCP implementation into believing the maximum bandwidth is less than the real one. This can be done by modifying internal TCP values such as window size on the transmission buffer. While this solution may be cleaner, it might require a lot of additional effort to implement and even require administrator rights, which is something we have tried to avoid when developing the application. Applications like Trickle can be a starting point for this future work [5].
Another -much easier- way to deal with this issue is to limit the packets that are being sent using *sleep* commands in our code. This seems to be a harsh solution for the issue, however, testing over dummy programs using network sniffers proved that this approach delivered a homogeneous limited bandwidth usage when using sleep iterations below 0.1 seconds. This solution also allowed to easily change the maximum speed limits from a user interface perspective.
Nevertheless, a major drawback of this solution is that it might be difficult to work with different level limits, such as in the case where you want to have different limits for each transfer and a global speed limit, as it would involve sharing data between threads.

14.2 Improved user interface

Currently, the program is easy to use but follows a design pattern different from the usual applications (it lacks an upper toolbar, there is no possibility to delete files from our list that are no longer used, does not provide a progress bar on file uploads and downloads...). This means the application could be classified on its latest beta stages, but still not ready to be released.
Most of the graphical interface was done keeping in mind the limited time available for the development, so the focus was to provide a solution even it was not the most efficient one. To solve this dearth, an additional development time of about a week or two would be required.

14.3 Designing a solution for non-PC machines

As good as Python is for our PC implementation, usually mobile devices such as smartphones and tablets do not come with a Python interpreter installed. This is an issue as these devices are becoming a huge market, and they are starting to pack hardware good enough to carry on encryption processes as well as mid-range desktop computers.
The best course of action for this limitation would be to develop different applications for each platform (at least the most used: Android, iOS and Windows Phone), since portable applications on mobile devices still suffer from efficiency issues, even if this fact is starting to change [6].
However, this task is neither easy or fast to do. This work would probably take up to a few months if a separate application for each platform was made.

14.4 Design a built-in file sharing feature

Making the most of the fact that every client generates an RSA key for themselves during the register process, a built-in sharing feature could be developed using those keys.
For instance, a user could ask the data server for a certain user's public key, and send them some of their file list, encrypted using their public key [Fig. 8]. This feature would provide a secure one-to-one file sharing, without needing to use the conventional non-automatic option.
There are many ways this feature could be implemented. Arguably the most efficient one would be to develop a new server whose sole purpose was to process file sharing requests. The method used by User2 to detect some file has been shared with him would be either programming a constant polling to the server handling file sharing, or to actually wait for incoming connections from that server. The second choice would probably outperform the first, making a more efficient use of the resources.

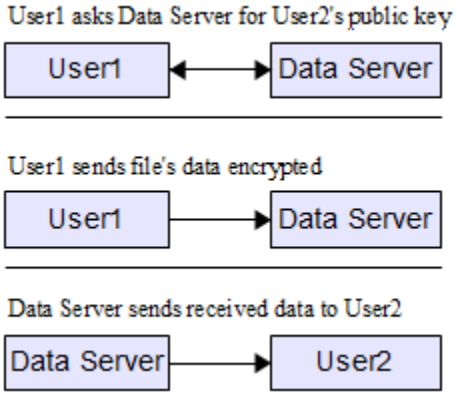

Fig 8: A possible implementation of an automatic file-sharing system.

14.5 Application auto-update feature

Witnessing the amount of possible future work available, a nice addition to the application would be to allow it to search for updates on startup.
This feature would probably require a redesign of many modules, but would greatly benefit the end user, suppressing the burden of re-downloading the entire application every time an update is released.
Adding the auto-update option would need the development of another server, deployed to inform client applications the release of any new version.

14.6 Restricting users' quota

If the application was ever to be released as a public domain on the Internet, it would probably crash within the first hours due users deliberately uploading files without control.
Establishing a limitation on the users' maximum quota is hard due the anonymous nature of the platform. However, some workarounds such as automatically deleting files that are not accessed during large periods of time could be made.

14.7 Developing an 'off-line' server

As it is currently designed, the application can't work without an Internet connection, since the client stores all its file list information on the server each time the application is closed. This allows users to access their account (and their files as a consequence) from multiple machines and locations. However, some users might be interested on ignoring this feature and be able to store the file list in their own computer, allowing them to use the basic share feature even without being connected to the Internet.

14.8 Improving file transfers considering brief interruptions on the connections

In rare occasions users might loose connection while uploading or downloading files to the File Server. On the current implementation, this causes these actions to fail, meaning the client has to start over again all the transfers from the beginning.

To prevent this from happening, additional data should be kept on the client, so that if a disconnect happens, the application knows which was the last successful packet sent. This functionality would require a *packet number* on the header, that should go encrypted along with the payload. On the File Server side, this will also require a redesign on the connection threads, implying the dead threads resulting from an unexpected disconnection did not close, but rather stay alive for a certain time waiting for a reconnection.

14.9 Enabling support for IPv6 and other upcoming technological standards

The application has been written to work under certain conditions met on the majority of actual systems. However, in the near future many of those conditions might change. A good example of this is the upcoming change to IPv6 addresses, which would cause our application to stop working. To fix this issue, many components of the application should detect the system's configuration and act accordingly to each machine's custom requirements.

14.10 Implementing a hardware-based encryption for Intel and AMD processors

Starting on early 2010, a hardware-based AES instruction set started to be featured on many Westemere Intel processors, later followed by AMD's Bulldozer family processors [7].

This instruction set extremely increases the performance on encryption algorithms using AES instructions, changing the computing burden from a software implementation to a hardware-based solution. Encryption algorithms using these features can go roughly 4x faster than their standard counterparts, as shown on Figure 9.

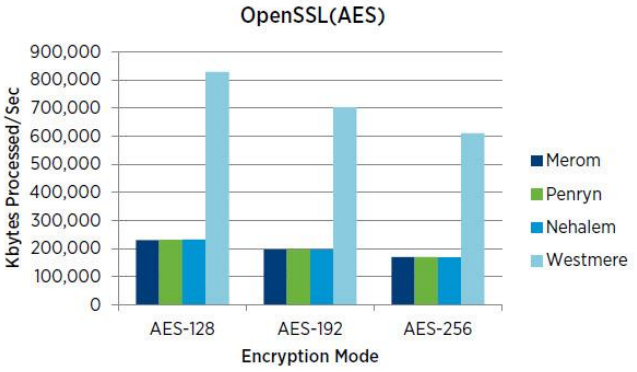

Fig 9. Performance increase on the hardware based AES implementation. Westmere is the first family of Intel processors featuring AES instructions. Source: http://www.derekseaman.com [8]

15 LICENSE

This document is released under the GNU Free Documentation License Version 1.3, 3 November 2008. A copy of this license can be found at

http://www.gnu.org/licenses/fdl-1.3.txt

REFERENCES

[1] Wassenaar Secretariat, Wassenaar Public Documentation February 11th, 2014 http://www.wassenaar.org/publicdocuments/index.html

[2] Bert-Jaap Koops, Crypto Law Survey, February 2013. http://www.cryptolaw.org/

[3] Multiple authors, Python Wiki Page, Article on GIL limitations, February 2nd, 2014 https://wiki.python.org/moin/GlobalInterpreterLock

[4] Nix Craft, "Linux tune network stack (buffers size) to increase networking performance", May 20th, 2009 http://www.cyberciti.biz/faq/linux-tcp-tuning/

[5] Marius Aamodt Eriksen, Trickle page on monkey.org, 2007 http://monkey.org/~marius/pages/?page=trickle

[6] Henning Heitkötter, Sebastian Hanschke and Tim A. Majchrzak, "Comparing cross-platform development approaches for mobile applications", 2012 http://www.academia.edu/3009209/Comparing_cross-platform_development_approaches_for_mobile_applications

[7] Jeffrey Rott, "Intel® Advanced Encryption Standard Instructions (AES-NI)", February 2nd, 2014. http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni

[8] Derek Seaman, Derek Seaman's Blog, September 8th, 2012 http://www.derekseaman.com/2012/09/how-much-does-evc-mode-matter-and-which.html