UNIVERSITÀ DI PISA

DEPARTMENT OF INFORMATION ENGINEERING

Bachelor's Degree in Computer Engineering

BACHELOR THESIS

IMPLEMENTATION OF A DISTRIBUTED PERSONAL INFORMATION RECOGNITION SYSTEM USING GRPC

Candidate:
Biagio Cornacchia

Supervisors:
Ing. Nicola Tonellotto
Ing. Carlo Vallati

Academic Year 2020-2021

INDEX

Chapter 1

Introduction

The adoption of AI technologies for data management is spreading more and more in recent years. Some companies are equipped with a system that aggregates a large amount of data relating to various sectors (for example the agri-food, one of the most important sectors of the economy): they collect scientific documents, tabular data in which there are scientific analysis, opinion polls, demographic research analysis and so on. All this information must be made available to the scientific community for analysis, studies, and statistics. However, within these documents there are a series of personal information (such as names, surnames, addresses etc.) that do not comply with the GDPR[1]'s regulations. The European **PIRA** *(Personal Information Recognition with AI)* project proposes to design and implement a new system for the secure management of sensitive data to allow data sharing among European countries, researchers, and organizations. It is necessary to find in these documents all those references to potentially private information that must be protected and anonymized. The goal is to ensure that the scientific value of the data remains intact, so the statistical power of the data is maintained.

To summarize, this project aims at:

- identify, classify, and link information related to personal data of sensitive nature (proper names, city names, identification codes, geographical coordinates) in textual documents

[1] The General Data Protection Regulation (GDPR) is a regulation in EU law on data protection and privacy in the European Union and European Economic Area. The GDPR's primary aim is to enhance individuals control and rights over their personal data and to simplify the regulatory environment for international business.

- effectively and efficiently anonymize the identified personal data, according to their nature, preserving the statistical value of the data unchanged.

Chapter 2

Technologies

To achieve this goal, the following technologies are:

1. **Microsoft Presidio** for data identification and anonymization
2. **gRPC** an open-source universal RPC framework used to encapsulate Microsoft Presidio modules so it could be integrated in other existing platform easily
3. **Docker** for easier deployment

2.1 Microsoft Presidio

Presidio is an open-source software written in Python which helps to ensure sensitive data is properly managed and governed. It provides fast identification and anonymization modules for private entities in text and images such as credit card numbers, names, locations, social security numbers, bitcoin wallets, US phone numbers, financial data and more. [1] In the figure below is shown an example of detection flow.

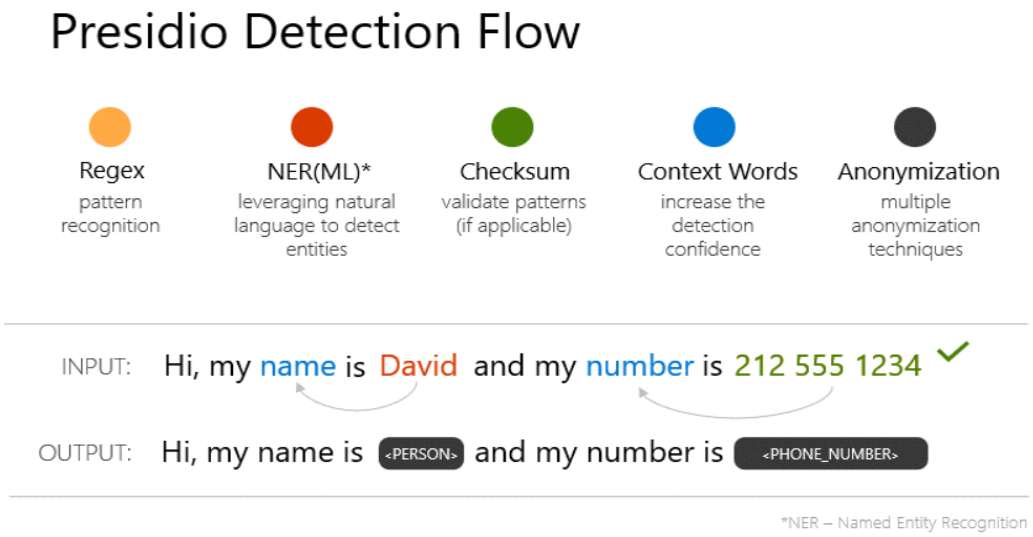

Figure 1: Microsoft Presidio detection flow

Microsoft Presidio consists of two independent modules:

1. the analyzer module, responsible for **identifying** sensitive information
2. the anonymizer module, responsible for the **obfuscation** of appropriate information

Presidio Analyzer is a python-based service for detecting PII *(Personally Identifiable Information)* entities in text. During analysis, it runs a set of different PII Recognizers, each one in charge of detecting one or more PII entities using different mechanisms as shown in Figure 2. It comes with a set of predefined recognizers, but it is also possible allow Presidio to be extended with new types of PII entities. There are different ways for extension:

1. deny-list based PII recognition - a list of words that should be found during text analysis
2. regex based PII recognition – the analyzer uses regular expressions to identify entities in text

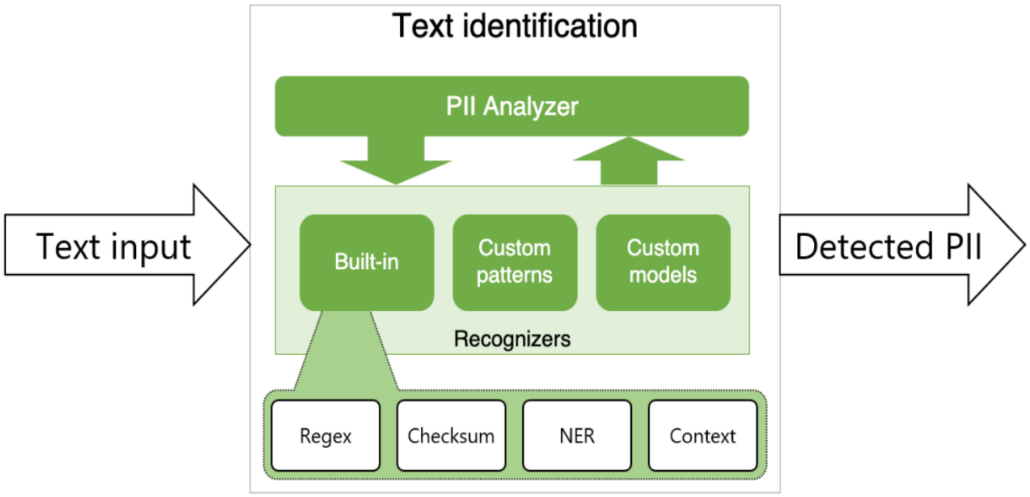

Figure 2: Analyzer process design

Presidio Anonymizer is a python-based module for anonymizing detected PII text entities with desired values. Presidio Anonymizer supports both anonymization and deanonymization by using operators. Operators are built-in

text manipulation classes which can be easily extended. In the figure below is shown the anonymizer process.

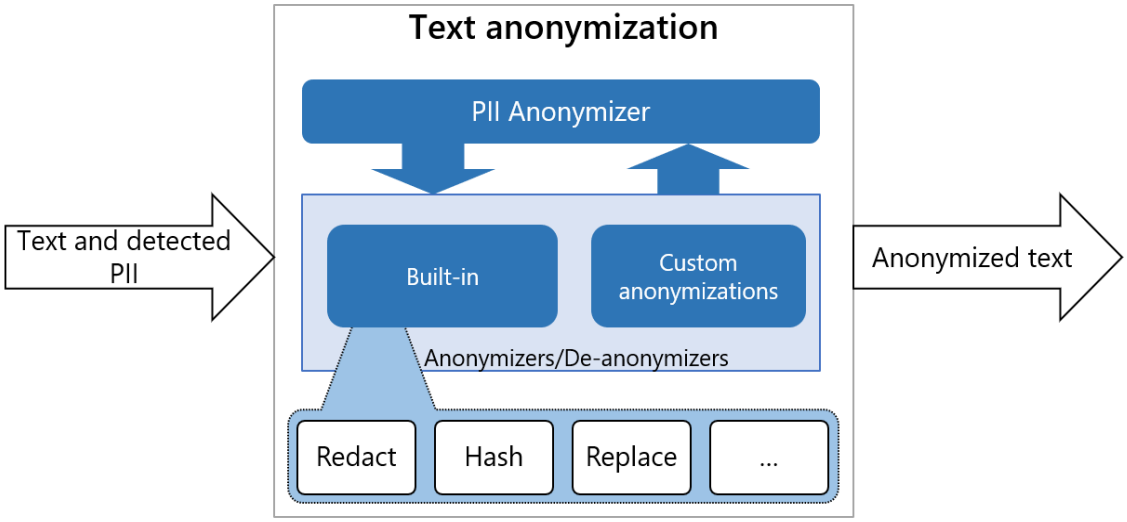

Figure 3: Anonymizer process design

It is important to note that Presidio is using trained ML models, therefore there is no guarantee that Presidio will find all sensitive information. Consequently, additional systems and protections should be employed. [1][2]

2.2 gRPC and Protocol Buffers

gRPC is a modern, open source, high-performance remote procedure call (RPC) framework to create modular applications. An important concept of gRPC is the idea of transparency: a client application can directly call a method on a server application on a different machine as if it were a local object whereas the details of the network communication must be hidden from the user. gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. On the server side, the server implements this interface and runs a gRPC server to handle client calls. On the client side, the client has a stub that provides the same methods as the server.

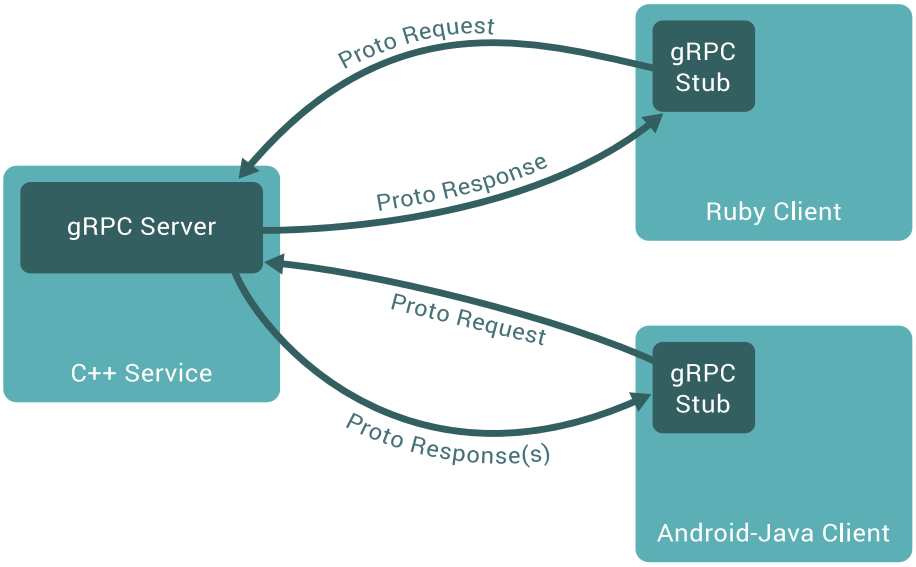

Figure 4: gRPC Scheme

In the figure above is shown a gRPC scheme in which clients and servers can run and talk to each other in a variety of environments and can be written in any languages supported by gRPC. [3]

By default, gRPC uses Protocol Buffers to encode data. It is a free and open-source cross-platform library used to serialize structured data and uses protoc with a special gRPC plugin to generate code from the proto file service definitions. [4]

2.3 Docker

Docker is an open platform for developing, shipping, and running applications. It offers many benefits besides this handy encapsulation, isolation, portability, and control. Installing on bare metal made the software painfully difficult to move around and difficult to update (two constraints that made it hard for IT to respond quickly to changes in business needs). Then virtualization came along. Docker uses OS-level virtualization to deliver software in packages called containers. Virtualization platforms (also known as “hypervisors”) allowed multiple virtual machines to share a single physical system, each virtual machine emulating the behaviour of an entire system, complete with its own operating system, storage, and I/O, in an isolated fashion. Virtual machines helped cut costs, because more VMs could be consolidated into fewer physical machines. Containers work like

VMs: they isolate a single application and its dependencies (all the external software libraries the app requires to run) both from the underlying operating system and from other containers. All the containerized apps share a single, common operating system. [5][6] The following figure shows a simple Docker overview.

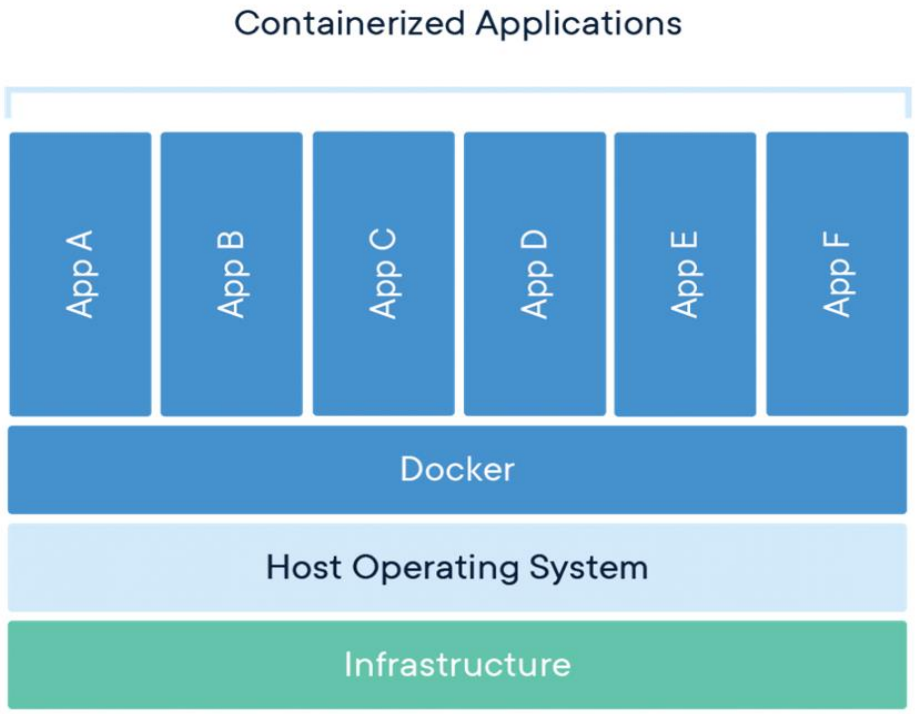

Figure 5: Docker overview

2.4 Design

As highlighted, the main tool to implement the proposed solution is Microsoft Presidio. The thesis activity consists in the creation of a Google Analyzer RPC server and a Google Anonymizer RPC server based on the Presidio source code. Then it is necessary to create a Google RPC server called *data loader* (for each module) that acts as a client for the corresponding server. In other words, it is a software module that reads the data from a local folder and sends it through Google Protocol Buffers to the Google RPC server of the respective module, receives the results and save them locally. Finally, each module was packaged with its Google RPC server and data loader as a Docker file.

Chapter 3

Project Architecture

The project is based on python and consists of two folders, one for the analyzer and one for the anonymizer. In each folder we can find the proto folder which contains a file called *model.proto* used to generate the needed stubs to create the gRPC client/server, the analyzer-temp/anonymizer-temp folder used by the server to save temporary files and four python files. For example, in the case of the analyzer:

1. analyzer_server.py – in which is defined the AnalyzerEntityServicer class used to implement the gRPC server.
2. analyzer_client.py – in which is defined the ClientEntity class used to implement the gRPC client.
3. data_loader.py – a python program to use the functionality of the analyzer.
4. clientGUI.py – a graphical user interface.

The analyzer-results folder and the anonymizer-results folder are used to contain the respective results generated by Microsoft Presidio modules.

3.1 Analyzer RPC Server

To perform the available operations of the analyzer, the gRPC server has been provided with four service methods defined in the proto file (figure 6).

```
service AnalyzerEntity {
    rpc sendFileToAnalyze(stream DataFile) returns (Ack);
    rpc sendEngineOptions(AnalyzerEngineOptions) returns (Ack);
    rpc sendOptions(AnalyzeOptions) returns (Ack);
    rpc getAnalyzerResults(Request) returns (stream AnalyzerResults);
}
```

Figure 6: Analyzer Service methods

1. sendFileToAnalyze

is used by the data loader to send the original text file that needs to be analyzed. Files will be divided into chunks. The server will assign an UUID that will be used during all the communication to identify uniquely the client information.

2. sendEngineOptions

The analyzer engine can be configured. Available options are:

- registry - an optional list of recognizers, that will be available instead of the predefined recognizers
- log_decision_process - defines whether the decision process within the analyzer should be logged or not
- default_score_threshold - minimum confidence value for detected entities to be returned
- supported_languages - list of possible languages this engine could be run on. Used for loading the right NLP models and recognizers for these languages

Using this method, the client eventually specifies analyzer engine options and sends them to the server. The server will store them into a json file and will return an acknowledgement message containing the UUID assigned from the server to the client during the first step.

3. sendOptions

The analyzer function can also be configured. Available options are:

- language - the language of the text
- entities - list of PII entities that should be looked for in the text. If entities value is "*None*" then all entities are looked for
- correlation_id - cross call ID for this request
- score_threshold - a minimum value for which to return an identified entity
- return_decision_process - it decides if the analysis decision process steps returned in the response

Using this method, the client specifies eventually his options and sends them to the server. The server will store them into a json file and returns and Ack message containing UUID assigned from the server to the client during the first step.

4. getAnalyzerResults

The client specifies his UUID and makes a request to get analyzer results. The server uses the original text and eventually json files containing the options specified by the client and performs the analysis. Then it returns found entities in the text, so the client will save them into a file called "filename-results.txt" which resides in analyzer-results folder.

Using the following command, we will generate the gRPC client and server interfaces from the proto service definition.

"python -m grpc_tools.protoc --proto_path=. ./proto/model.proto --python_out=. --grpc_python_out=."

The **AnalyzerEntity** class defines the methods exposed by the gRPC server whereas the **AnalyzerEntityServicer** class implements them into the file analyzer_server.py.

3.2 Analyzer RPC Client

To implement the client side, a class called **ClientEntity** was created and defined in the file analyzer_client.py. This class has some methods used by the client to setup the analyzer configuration and to send a request to start the analysis.

1. **setupDenyList** is used to setup a deny list. It has two arguments:
   - a list of the supported entity
   - a list of words to detect
2. **setupRegex** is used to set a regex pattern. It has three arguments:
   - the entity supported by this recognizer
   - one or more patterns that define the recognizer
   - list of context words to help detection
3. **setupOptions** is used to setup all the other options specifying the right options file and returns an integer.
4. In the end, to perform analysis there is a function called **sendRequestAnalyze**. This function takes an argument (a filename) and (after a check for the required files) sends the original text file (divided into chunks of 1MB) and eventually the analyzer engine and the analyzer function configuration. Then makes a request to get the analyzer result. It returns an integer:
   - if some required files do not exist or the request for the analyzer results fails returns -1
   - if there is a gRPC exception such as 'server unavailable' returns -2
   - if some required files were not received correctly by the server return 0
   - if the operation was successful returns 1

3.3 Analyzer Sequence Diagram

This sequence diagram shows the logic behind the actors and the system in performing the task.

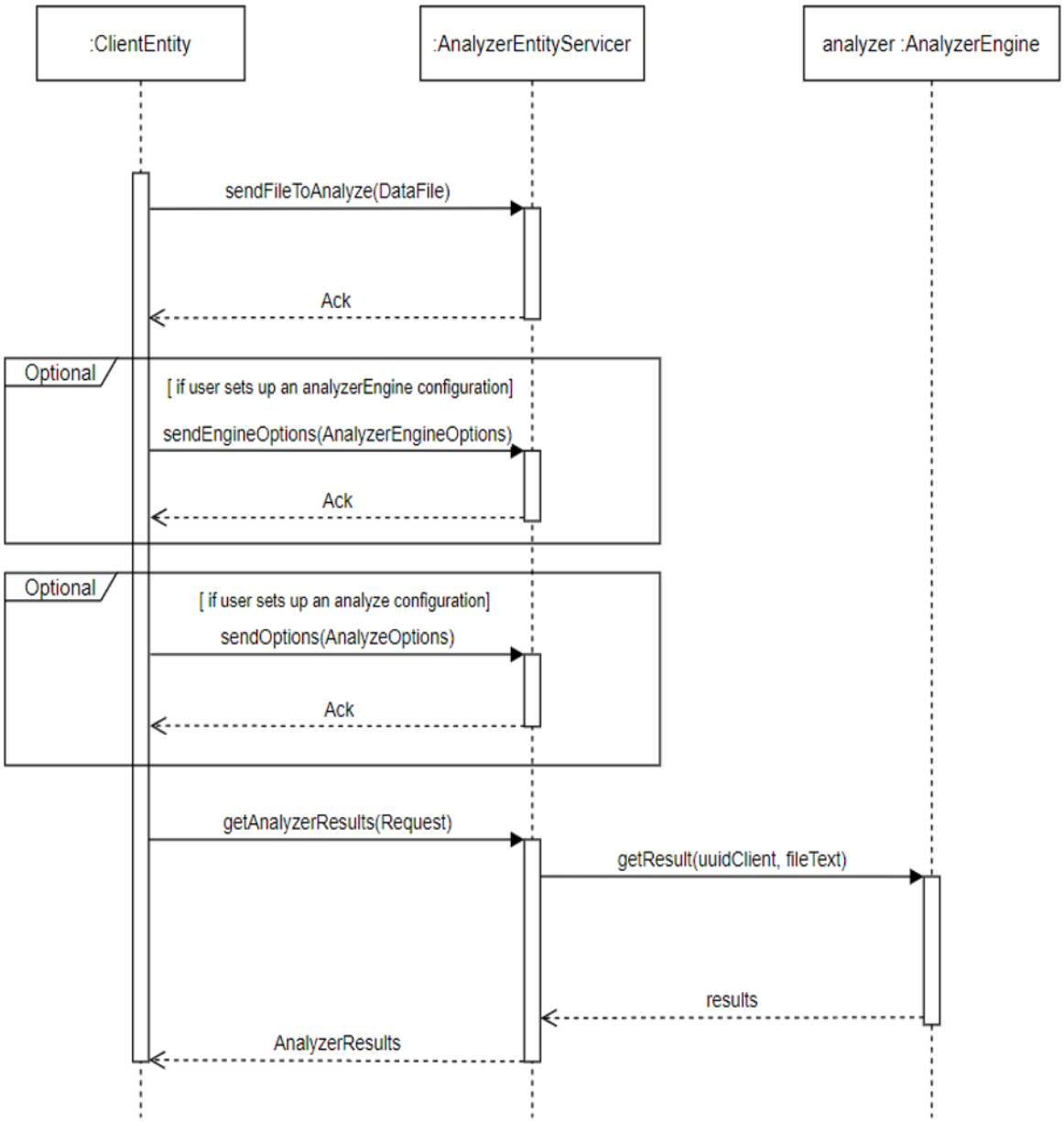

Figure 7: Analyzer Sequence Diagram

To perform a simple file analysis operation, the client entity calls the sendFileToAnalyze function specifying the file (s) to be analyzed contained in the local files folder. The Entity Servicer Analyzer responds to this request with an

acknowledgment message which reports the number of chunks that the server has received. By examining this data, the client can make sure that the file has been sent successfully to the server and continue with the analysis. The second step is to look for any specified options to apply during the analysis. For example, the client might wish to search within the text only a certain entity or wants to use a custom identifier (specifying a pattern regex or a deny list for identification). This step is optional as the client could perform the analysis using the default settings of Microsoft Presidio. In this case, the ACK message will contain the UUID that the server has assigned to that client. The last step is to start the analysis and request the results produced from the server. This is initiated by calling the getAnalyzerResult function (specifying an UUID). At this point the server runs the monitoring analyzer module and sends the results to the client, which will be saved in a local text file.

3.4 Anonymizer RPC Server

For the server-side it is necessary to introduce the Anonymizer RPC Service defined in the proto file. To perform anonymization six service methods has been created.

```
service AnonymizerEntity {
    rpc sendRecognizerResults(stream RecognizerResult) returns (FileAck);
    rpc sendAnonymizedItems(stream AnonymizedItem) returns (FileAck);
    rpc sendConfig(Config) returns (FileAck);
    rpc sendFile(stream DataFile) returns (FileAck);
    rpc getText(Request) returns (stream DataFile);
    rpc getItems(Request) returns (stream Item);
}
```

Figure 8: Anonymizer Service methods

1. sendRecognizerResults

is used by the data loader to send the found entities in the text. RecognizerResult, shown in Figure 9, is an exact copy of the RecognizerResult object from presidio-analyzer. Indeed, the message contains the start location of the detected entity, the end location of the detected entity, the score of the detection and the type of the entity.

```
message RecognizerResult {
    int32 start = 1;
    int32 end = 2;
    float score = 3;
    string entity_type = 4;
    string uuidClient = 5;
}
```

Figure 9: RecognizerResult Message

2. sendAnonymizedItems

is used by the data loader to send a list of information about the anonymized entities to perform deanonymization. In this case the message contains the start index of the changed text, the end index of the changed text and the type of the entity.

```
message AnonymizedItem {
    int32 start = 1;
    int32 end = 2;
    string entity_type = 3;
    string operator = 4;
    string uuidClient = 5;
}
```

Figure 10: AnonymizedItem Message

3. sendConfig

is used to send the operator configuration. Possible operators for the anonymizer are:

- encrypt - anonymize the text with an encrypted text using AES
- replace - replaces the PII text entity with a new string
- redact - redact the string - empty value
- mask - mask a given amount of text with a given character
- hash - hash given text with sha256/sha512/md5 algorithm

The configuration file is called *operatorConfigAnonymizer* and resides in the config folder.

Instead for deanonymization is supported only one operator:

- decrypt - decrypt text to from its encrypted form using the key supplied by the user for the encryption

The configuration file is called *operatorConfigDeanonymizer* and resides in the config folder.

4. sendFile

is used by the data loader to send the original text file that needs to be anonymized. Files will be divided into chunks. The server will assign a UUID that will be used during all the communication to identify uniquely the client information.

5. getText

The client specifies his UUID and specifies the type of request (anonymization or deanonymization). For anonymization the result will be saved into a file called "filename-anonymized.txt" which resides in the anonymized-results folder. Instead, for deanonymization the result will be saved into a file called "filename-deanoymized.txt" that resides in the same folder.

6. getItems

The client specifies his UUID and makes a request to get items. Anonymize() and Deanonymize() function returns the anonymized text and a list of items that contains information about the anonymized/deanonymized entities. For anonymization the result will be saved into a file called "*filename-anonymized-items.txt*" which is contained in the anonymized-results folder. Instead, for deanonymization the result will be saved into a file called "filename-deanonymized-items.txt" which is contained in the same folder.

3.5 Anonymizer RPC Client

To perform anonymization/deanonymization there are four functions.

1. sendRequestAnonymize

This function takes an argument (a filename) and (after a check of the required files) sends the original text file, the analyzer results and eventually a configuration file. Then makes a request for the anonymized text and anonymized items. It returns an integer:

- if some required files do not exist or the request for text/items fails returns -1
- if there is a gRPC exception such as 'server unavailable' returns -2
- if some required files were not received correctly by the server return 0
- if the operation was successful returns 1

2. sendRequestDeanonymize

This function takes an argument (a filename) and (after a check of the required files) sends the anonymized text file, the anonymizer results, and a configuration file (in this case is required because you must specify a key for decrypt). Then makes a request for the deanonymized text and deanonymized items. It returns an integer:

- if some required files do not exist returns -1
- if there is a gRPC exception such as 'server unavailable' returns -2
- if some required files were not received correctly by the server return 0
- if the operation was successful returns 1

3. sendRequestForText

This function takes three arguments (a filename, uuidClient assigned by the server and a requestType that can be 'anonymize' or 'deanonymize') and sends a request to get anonymized or deanonymized text. It returns an integer:

- if the request for the anonymized/deanonymized text fails returns -1

- if the operation was successful returns 1

4. sendRequestForItems

This function takes three arguments (a filename, uuidClient assigned by the server and a requestType that can be 'anonymize' or 'deanonymize') and sends a request to get anonymized or deanonymized items. It returns an integer:

- if the request for the anonymized/deanonymized items fails returns -1
- if the operation was successful returns 1

3.6 Anonymizer Sequence Diagram

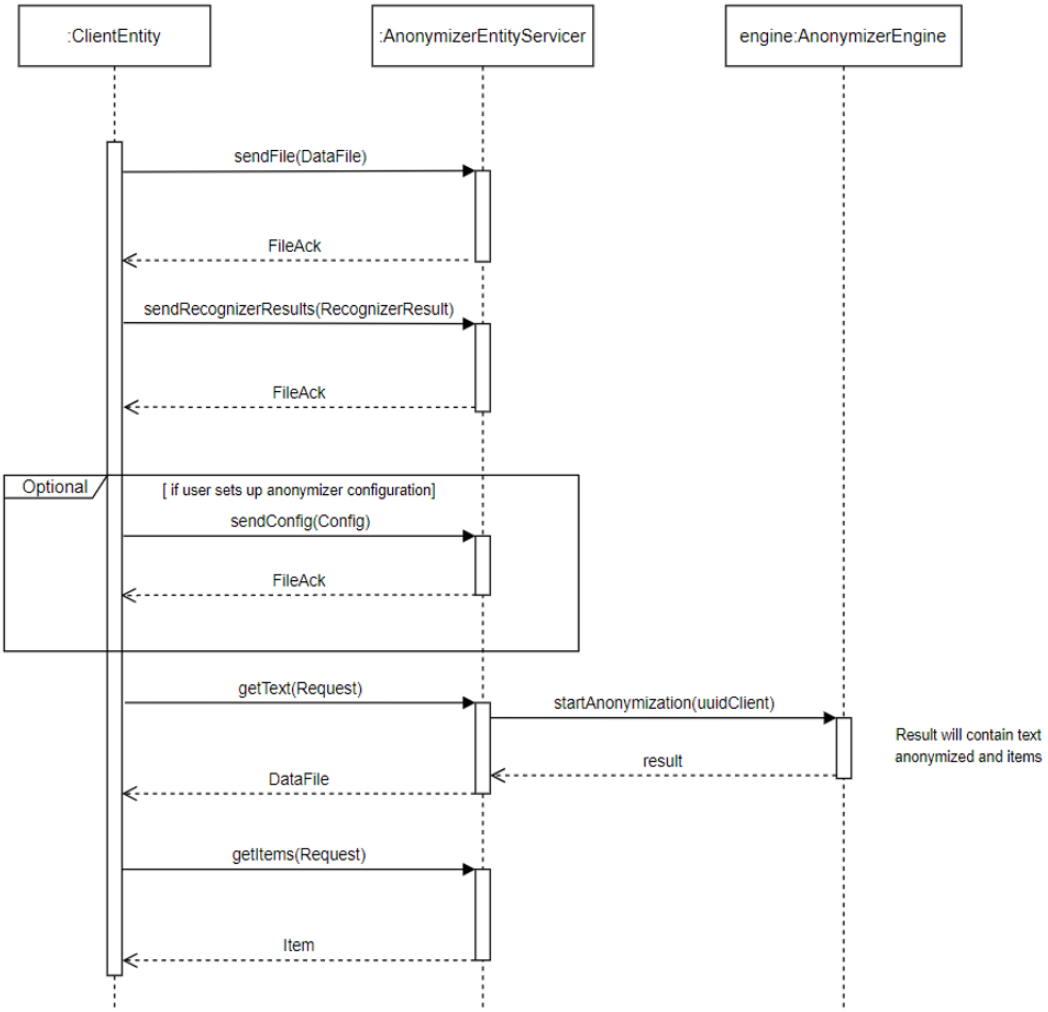

Figure 11: Anonymizer Sequence Diagram

To perform anonymization as shown in Figure 11, the client entity calls the sendFile function specifying the file (s) to be anonymized contained in the local files folder. The Entity Servicer Anonymizer responds to this request with an acknowledgment message which reports the number of chunks that the server has received. By examining this data, the client can make sure that the file has been sent successfully to the server and continue with the operation. The second step is to look for any specified options. For example, the client might want to use a specific operator (hash, replace, encrypt etc.) to anonymize a specific entity type (for example PERSON, LOCATION and so on). This step is optional as the client

could perform the anonymization using the default settings. Then the last step is to send a request to get the anonymized text. This is initiated by calling the getText function (specifying an UUID). At this point the server runs the anonymizer module and sends the text to the client. The client can also ask for an items file which will contain more detailed information about the anonymized entities.

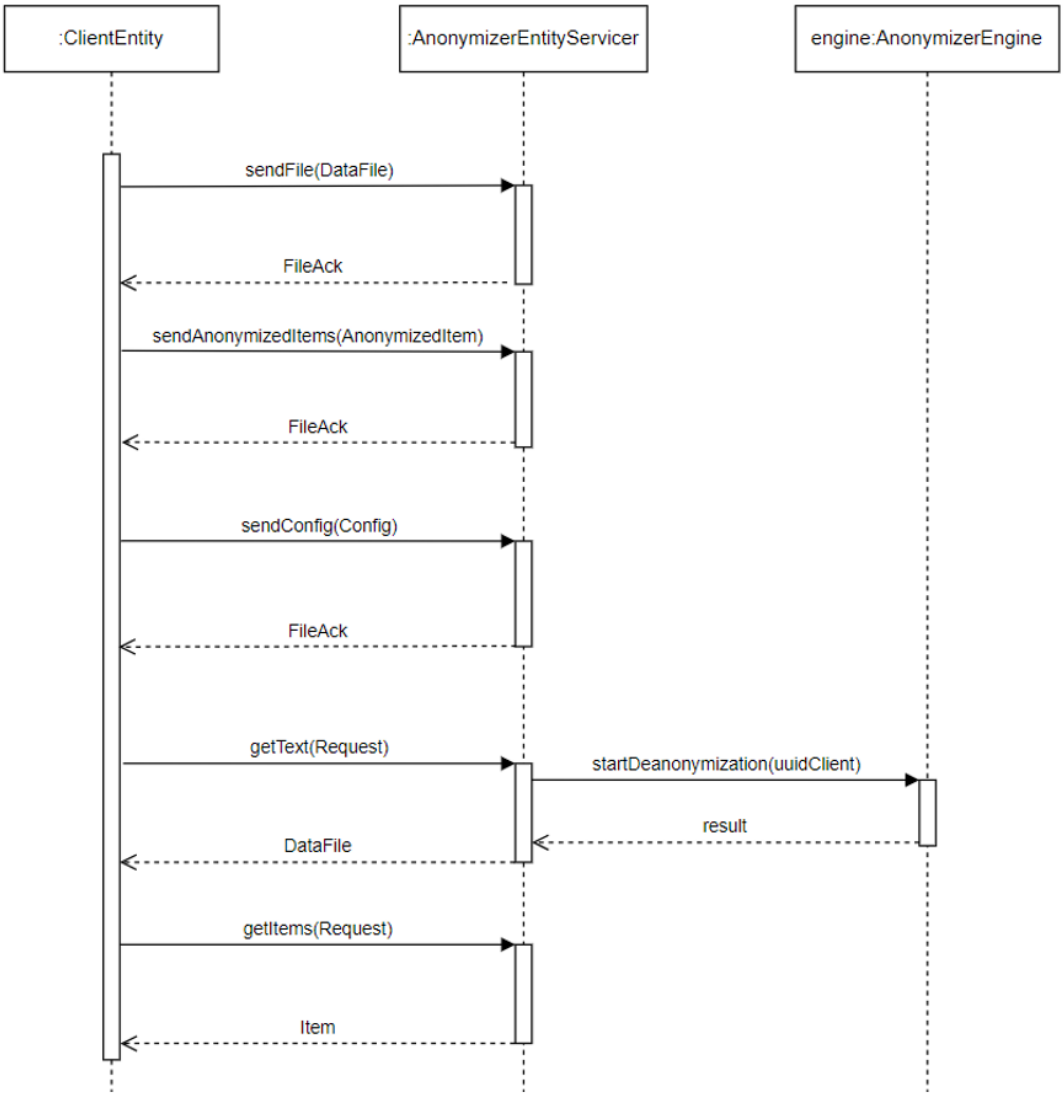

Figure 12: Deanonymizer Sequence Diagram

Figure 12 shows the deanonymization process. The client entity acts in the same way as for the anonymizer. In this case only the second step is different. A configuration file must exist. The client must specify the key that will be used to decrypt entities encrypted with the same AES key.

Chapter 4

GUI

To make this software easier to configure and to make the expected outputs clearer to read, a graphical interface has been created. For this purpose, **tkinter** is the library that allowed me to create the graphical user interfaces (GUI) using Python. [7]

4.1 Example of analysis

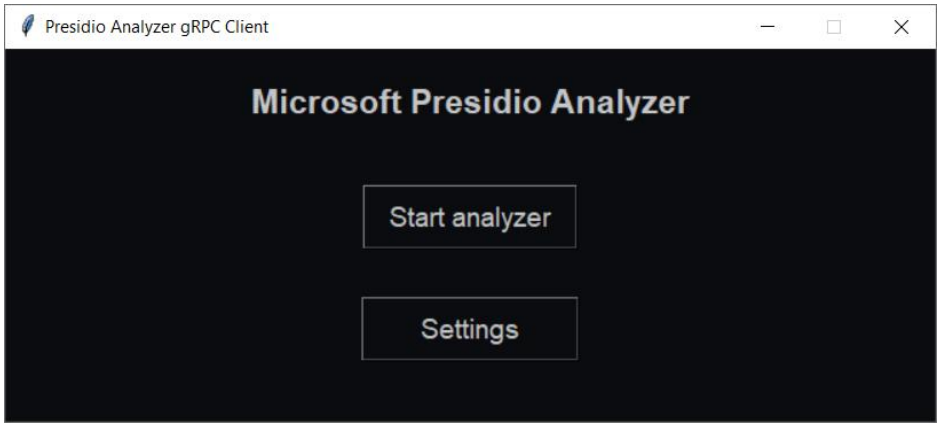

Figure 13: Analyzer Client GUI

Through the interface (figure 13) it is possible to configure the settings such as the analyzer server data or specify the options to be applied during the analysis or it is also possible to proceed directly with the file analysis using a default configuration. Here are a few examples sentences currently supported by Presidio:

```
Hello, my name is David Johnson and I live in Maine.
My credit card number is 4095-2609-9393-4932 and my crypto
wallet id is 16Yeky6GMjeNkAiNcBY7ZhrLoMSgg1BoyZ.
```

Performing the analysis, the following results will be obtained:

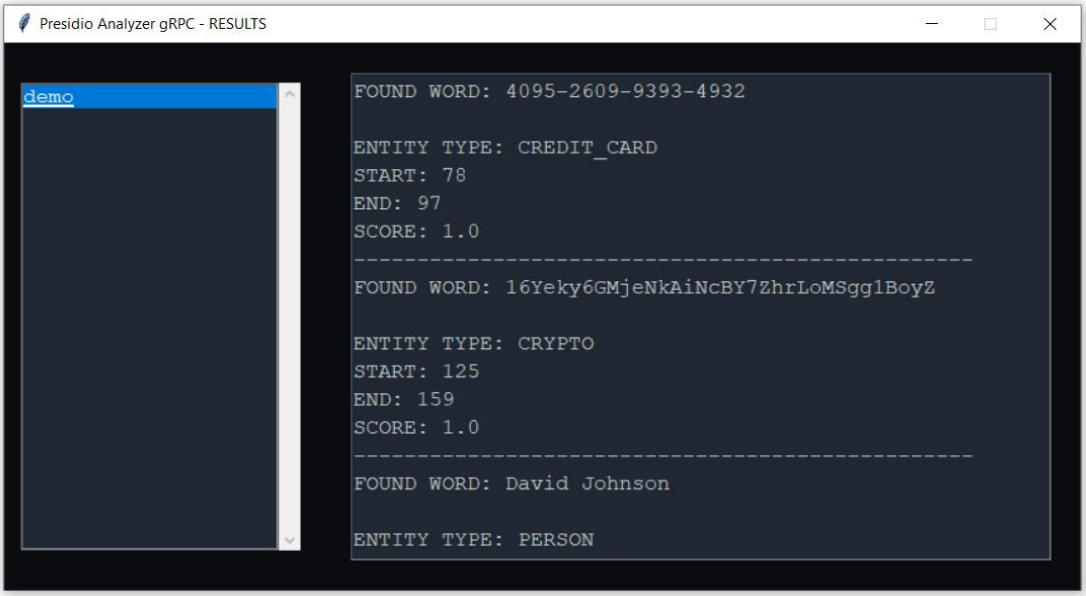

Figure 14: Analyzer output

We can see how the analyzer was able to recognize a personal name (David Johnson), his location (Maine), his credit card number and his wallet id. Other information are indexes that indicate the position of the beginning of the entity and the end of the entity within the text and the score which indicates the accuracy of the result.

4.2 Example of anonymization/deanonymization

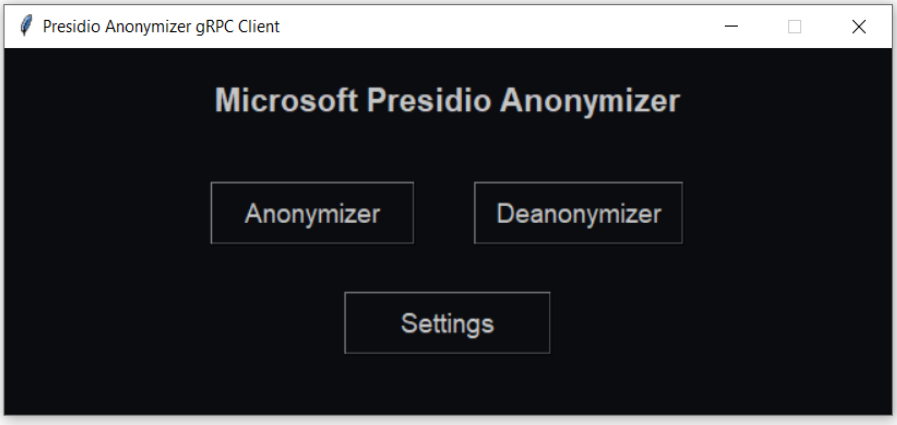

Figure 15: Anonymizer Client GUI

As with the analyzer, this interface (figure 15) allows you to enter the server information and perform an anonymizer configuration (specifying for a certain type of entity and it is possible to choose between various anonymizers). For

example, you might encrypt all personal names while using a “replace” type operator for all other entities. Taking as an example the same text as before and using an "encrypt" operator for personal names and a "replace" operator for all other entries (as default) we will have the following output:

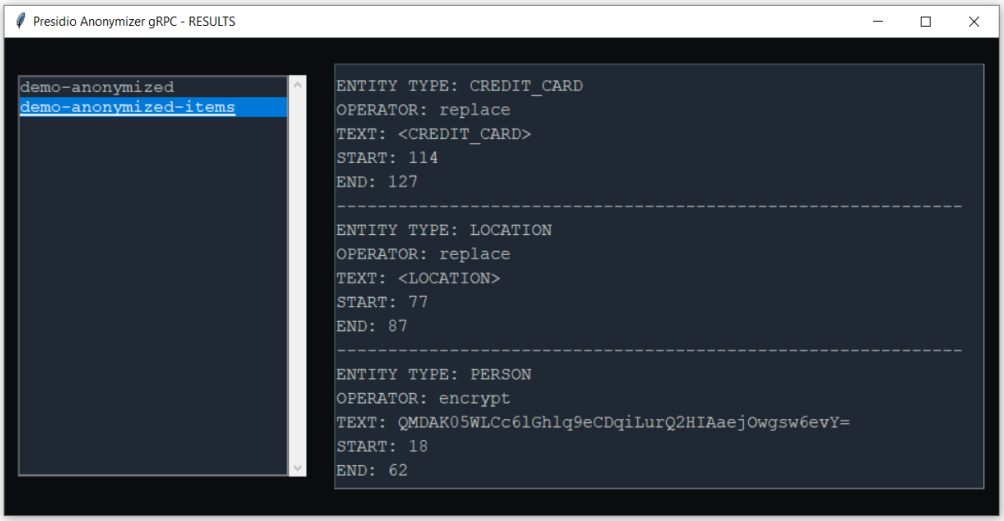

Figure 16: Anonymized items

The anonymized text instead will be:

`Hello, my name is` **`QMDAK05WLCc6lGhlq9eCDqiLurQ2HIAaejOwgsw6evY=`** `and I live in` **`<LOCATION>.`**

`My credit card number is` **`<CREDIT_CARD>`** `and my crypto wallet id is` **`<CRYPTO>.`**

This operation, as you can see on the left of the interface, generated two types of output:

- an output containing the anonymized text
- an output containing the information on the entities and the respective anonymizers that have been applied

This information will be used in a possible de-anonymization phase where it will be necessary to specify the AES key used in the anonymization phase. In the figure below there is an example of a configuration to be applied if the anonymized file contains encrypted *PERSON* type entities (obviously you will need to use the same key used to encrypt).

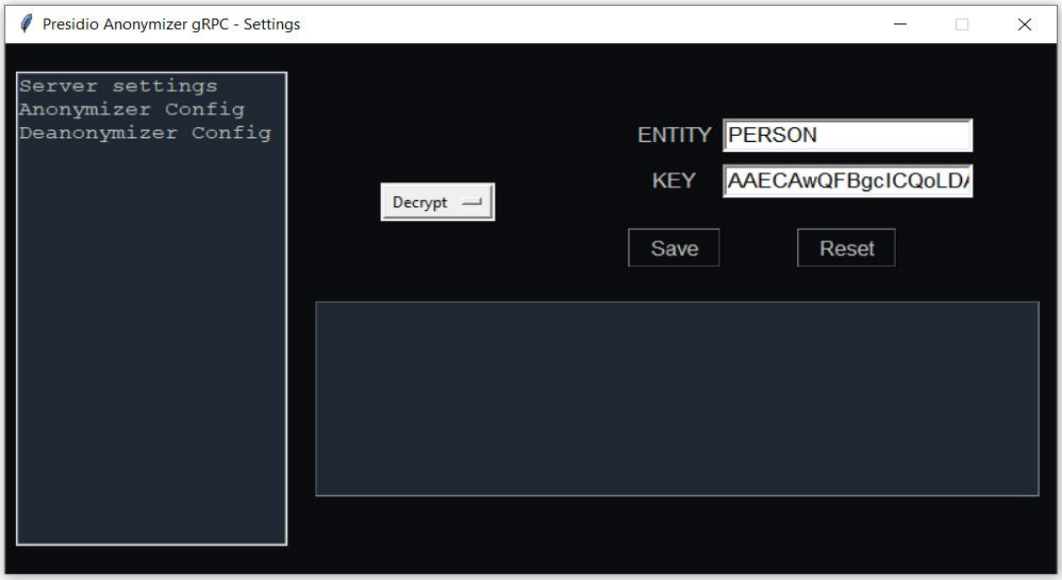

Figure 17: Deanonymizer settings

Therefore, by starting the deanonymizer and specifying the *filename-anonymized.txt* file generated by the anonymizer as the input file, the following outputs will be obtained:

```
Hello, my name is David Johnson and I live in <LOCATION>.
My credit card number is <CREDIT_CARD> and my crypto wallet
id is <CRYPTO>.
```

Deanonymized-items.txt will be:

```
ENTITY TYPE: NUMBER
OPERATOR: decrypt
TEXT: David Johnson
START: 18
END: 31
```

Chapter 5

Conclusions

The work carried out made it possible to create a fully-fledged system capable of offering a solution to many companies that have large amounts of data but which, without the right attentions, cannot be made available to communities for statistical purposes or for carry out studies. The potential made available by software such as Microsoft Presidio has given a turning point to this problem allowing the creation of simple python modules able to recognize and consequently obfuscate the various personal information of different nature present in text files, exploiting a machine learning system for such operations. However, it must be considered that in some cases there may be errors during the recognition of a particular entity, but you can improve the recognition by adding new identification systems such as deny-lists or regex patterns with specific context words to increase accuracy. The software created also lends itself very well to extensions and customizations and it can be directed towards a more specific business. At the same time what made its implementation efficient is gRPC which offers a “refresh” of the old RPC design method using such technologies as HTTP/2 and Protocol Buffers. In fact, Protocol Buffers made it possible to define in a very simple way the structure of the messages exchanged in the client-server communication and making sure that they can be read with facility and precision.

BIBLIOGRAPHICAL REFERENCES

[1] https://microsoft.github.io/presidio/

[2] https://github.com/microsoft/presidio

[3] https://grpc.io/docs/what-is-grpc/introduction/

[4] https://developers.google.com/protocol-buffers/docs/reference/python-generated

[5] https://docs.docker.com/get-started/overview/

[6] https://www.infoworld.com/article/3310941/why-you-should-use-docker-and-containers.html

[7] https://docs.python.org/3/library/tkinter.html