MÁSTER UNIVERSITARIO EN CIBERSEGURIDAD

Creación de bot en Twitter

TRABAJO FIN DE MÁSTER

Autor: Eduardo Graván Serrano

Director: Manuel Sánchez Rubio

Universidad de Alcalá

2021

Escuela de Posgrado

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN CIBERSEGURIDAD

Trabajo Fin de Máster
Creación de bot en Twitter

Autor: Eduardo Graván Serrano

Tutor/es: Manuel Sánchez Rubio

TRIBUNAL:

Presidente: << Nombre y Apellidos >>

Vocal 1º: << Nombre y Apellidos >>

Vocal 2º: << Nombre y Apellidos >>

FECHA: << Fecha de depósito >>

Agradecimientos

En primer lugar, me gustaría dedicar unas palabras de agradecimiento a mis padres. A lo largo de toda mi vida han luchado por mi educación y sé que mi graduación les llenará de orgullo tanto como a mí.

Me gustaría agradecer también a Manuel, mi tutor, por haberme guiado a lo largo de los meses de desarrollo de todo el proyecto.

Gracias a todos los amigos y familiares que he usado de soporte a la hora de debugear el código a lo largo de todo el desarrollo, aunque en muchas ocasiones no entendiesen nada de lo que estaba hablando.

Por último, me gustaría dar las gracias a la comunidad de desarrolladores de Android y en especial a *StackOverflow* por ser un pilar fundamental a la hora de resolver dudas y aportar ejemplos para ayudarme a construir las bases del proyecto.

Índice

Índice de Imágenes

Índice de Tablas

Resumen

Este Trabajo Fin de Máster está separado en dos partes fundamentales, ambas relacionadas con la red social y plataforma conocida como Twitter.

La primera de ellas consiste en el desarrollo de una aplicación de escritorio con el objetivo de dar una interfaz de acceso a Twitter a través de la API oficial que la plataforma pone al alcance de los desarrolladores, con el objetivo de solventar la necesidad de un usuario especializado de esta plataforma de cara a realizar búsquedas y guardar los resultados para su posterior análisis.

La segunda parte de este proyecto consiste en el estudio del funcionamiento de las redes de bots o *botnets* en la plataforma Twitter, culminando con el desarrollo de una pequeña prueba de concepto de una red de bots que cumpla con los requisitos para ser considerada como tal.

Palabras Clave

Twitter, Python, Social Media, Botnet, Social media scraping and automation

Abstract

This Master's Thesis is separated into two fundamental parts, both related to the social media platform known as Twitter.

The first part consists of the development of a desktop application with the objective of providing an interface to access Twitter through the official API that the platform makes available to developers, with the aim of solving the need of a specialized user of the platform to carry out searches and save the results for later analysis.

The second part of this project consists of the study of the operation of botnets on the Twitter platform, culminating in the development of a small proof of concept of a botnet that meets the requirements to be considered as such.

Key Words

Twitter, Python, Social Media, Botnet, Social media scraping and automation

1. Introducción

Según estudios publicados en julio de 2021 por Global WebIndex [1], un total de 4.48 billones de personas globalmente cuentan con cuentas de usuario activas en algún tipo de red social, lo que supondría un total del 56.8% de la población mundial.

A estas impresionantes cifras se debe añadir el dato de que este número no para de incrementarse cada año, registrándose la diferencia de usuarios anual entre 2020 y 2021 en un incremento del 13.1%, esto es, se han sumado 520 millones de nuevos usuarios a las redes sociales.

Entre estas redes sociales, una de las más conocidas y utilizadas globalmente es Twitter [2], que también ve un crecimiento anual en su número de usuarios activos mensualmente, situándose cerca de los 353 millones de usuarios.

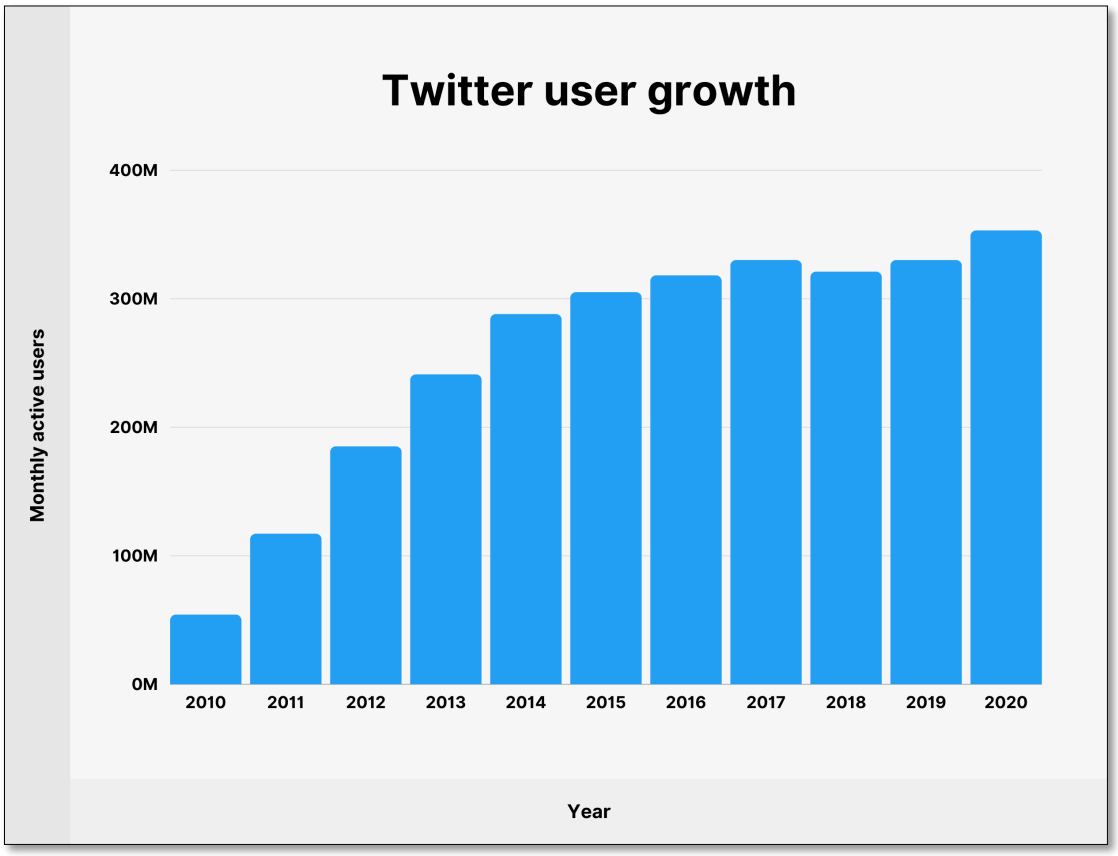

Figura 1. Incremento de usuarios mensuales en Twitter [3].

Este tipo de redes sociales es utilizado por un gran número de usuarios para publicar sus opiniones sobre temas sin mayor relevancia como pueden ser resultados deportivos, música, o cualquier otro tipo de contenido multimedia de entretenimiento. Pero, además, estas redes sociales son utilizadas activamente para compartir opiniones sobre temas mucho más profundos como puede ser la política, religión, etc., y, al ser utilizadas por un número de usuarios tan masivo, este tipo de interacciones a través de las redes sociales tiene un gran peso en el ideario y pensamiento de la sociedad actual [4].

Por todo esto, se cree que el estudio y análisis de los contenidos subidos a redes sociales tiene un gran interés académico, social e incluso financiero a través de campañas de publicidad dirigidas.

Debido a la ingente cantidad de datos que se almacenan en este tipo de redes sociales, recuperar esta información para su estudio de forma manual es impracticable, y por ello surge la necesidad de crear herramientas de automatización en cuanto a la búsqueda y recuperación de estos datos desde las redes sociales.

Además, las corrientes de opinión en este tipo de plataformas podrían ser fácilmente manipuladas a través de la creación de redes de usuarios falsos controlados por programas automatizados [5] (bots), que se encargasen de apoyar ciertas causas de diversa índole en redes sociales, desvirtuando la percepción de cualquier usuario sobre la opinión real de la población frente a un determinado tema, y pudiendo incluso influir en su propia opinión.

El trabajo descrito en este documento se centrará en estos dos últimos puntos descritos, enfocándose en la red social Twitter, esto es:

- Se creará una herramienta que permita hacer de interfaz con la red social, permitiendo lanzar búsquedas personalizadas, y permitiendo exportar el resultado de estas búsquedas para su posterior análisis.

- Se estudiará el funcionamiento de las redes de bots en Twitter, creando una pequeña prueba de concepto que cumpla con las funciones básicas de este tipo de redes.

2. Objetivo

Al ser un trabajo dividido en dos partes fundamentales, el objetivo de este trabajo también está dividido en el desarrollo de dos sistemas independientes.

El primer objetivo es el desarrollo de una aplicación que permita a un usuario de la red social Twitter realizar búsquedas especializadas en la plataforma a través de la API de desarrolladores oficial proporcionada por Twitter. Este sistema contará, al menos, con las siguientes funcionalidades:

- Posibilidad de realizar búsquedas en la red social Twitter.
- Posibilidad de crear listas de seguimiento de usuarios sobre los que realizar búsquedas en la red social.
- Posibilidad de exportar los datos para su posterior análisis en programas externos.

Para conseguir esto, se ha hecho uso del lenguaje de programación Python, y más en concreto, de la librería Tweepy [6], la cual posibilita el acceso a la API de desarrolladores [7] de Twitter de una forma rápida y sencilla. Además, uno de los requisitos principales de esta aplicación era posibilitar al usuario hacer un seguimiento de una serie de cuentas en Twitter, para poder dirigir sus búsquedas a aquellas cuentas que le supongan un mayor interés durante su uso de la aplicación desarrollada; para ello, se ha creado una base de datos encargada de almacenar esta información.

El segundo objetivo de este proyecto es la creación de una prueba de concepto de una *botnet* en la red social Twitter, con el objetivo principal de comprender como se podría crear una red de estas características con fines políticos y/o económicos en un entorno real, y demostrar lo sencillo que es desarrollar y desplegar una red que cumpla una serie de funciones básicas, a pesar de las restricciones que la plataforma intenta imponer frente a este tipo de acceso automatizado a la plataforma. La *botnet* deberá tener, al menos, las siguientes características:

- Tener una infraestructura cliente-servidor, con un servidor de *Command & Control* (C2) encargado de enviar instrucciones a los bots (clientes).
- Los bots deberán ser capaces de iniciar sesión en Twitter y ejecutar los comandos que reciban del servidor C2 de forma totalmente autónoma y automática.
- Como mínimo, los bots deberán ser capaces de dar “Me Gusta” a los Tweets que cumplan cierto criterio preestablecido.

Para conseguir estos objetivos, se deberá hacer un estudio inicial del estado del arte en cuanto a la red social Twitter en sí misma, su reglamento en cuanto a acceso automatizado, y las distintas soluciones existentes actualmente para acceder a los datos de la plataforma de una forma automatizada.

Con esto, se pretenden construir dos sistemas independientes: uno que posibilite el acceso a la red social a través de una interfaz gráfica de usuario, con la posibilidad de guardar los resultados de búsqueda para su posterior análisis; y un segundo sistema que consista en una *botnet* capaz de acceder y ejecutar ciertas acciones de forma totalmente automática en la plataforma.

3. Estado del arte

En este apartado se hará un estudio de las tecnologías y herramientas disponibles para crear estas soluciones para recuperar datos de la red social Twitter de una forma automatizada.

3.1. Twitter

Twitter [8] es una red social y plataforma que permite enviar mensajes cortos, conocidos como tweets, de hasta 280 caracteres. Se permite también subir contenido multimedia como imágenes, vídeos, audios, etc., así como incluir enlaces a otras páginas web en estos tweets.

Figura 2. Logo de Twitter.

Todos los mensajes publicados en Twitter son públicos por defecto, pero los usuarios pueden decidir proteger su cuenta, haciendo que sus tweets solo sean accesibles para aquellos usuarios que le siguen. Para seguir a un usuario con la cuenta protegida, se requiere una confirmación expresa por el usuario para poder seguir a la cuenta.

A esta funcionalidad se le suma la capacidad de responder a dichos mensajes. Cualquier usuario puede responder a un tweet público con su propio tweet, quedando el último listado debajo del tweet original. Esto permite crear hilos de comentarios, posibilitando tener conversaciones y/o debates sobre los temas publicados en la plataforma.

Además, se permite a los usuarios marcar los tweets de dos formas distintas.

Twitter permite darle “me gusta” a cualquier tweet, lo que lo añadirá a una lista de tweets gustados en el perfil de la persona que ha dado click al botón. Esta información es pública, y para cada tweet se puede ver el número de “likes” que ha recibido, así como un listado de las cuentas que han realizado esta acción (siempre que no sean cuentas privadas).

De la misma manera, se puede dar “retweet” a un tweet. El funcionamiento en cuanto a visibilidad es exactamente el mismo al de los “me gusta”, cualquier usuario puede ver que usuarios han dado

retweet al tweet, así como ver el número total de retweets que tiene un tweet. Además, la persona que da retweet a un tweet, comparte el tweet original en su perfil.

Adicionalmente, en el año 2015, se añadió una nueva forma de hacer retweet, que es el citado de tweets. Cuando se da retweet a un determinado mensaje, se puede escoger entre hacer un retweet normal con la funcionalidad explicada anteriormente, o citar el tweet. Al citar un tweet, el usuario que decide citar podrá escribir un tweet nuevo, quedando el tweet original en un pequeño recuadro debajo del nuevo tweet, enlazando el tweet nuevo con el original.

Como último punto a remarcar sobre la publicación de tweets, se debe decir que los tweets pueden incluir una cadena de caracteres precedido por un carácter especial que permite que se añadan a una lista de tweets relacionados. El carácter especial es **#**, y la cadena completa formada por *#cadena*, se conoce como ***hashtag***.

Los hashtags permiten a los usuarios publicar tweets que serán agrupados bajo ese identificador, permitiendo crear un hilo de mensajes relacionados con el mismo tema, permitiendo darle visibilidad en medida del número de gente que esté publicando tweets bajo ese hashtag.

Twitter organiza estos hashtags en función del número de atención que estén recibiendo por parte de sus usuarios en un determinado momento, permitiendo saber cuales son los temas "del momento" en la plataforma. En el momento en el que se está escribiendo esta memoria, los hashtags más importantes en España son los que aparecen en la figura 3.

Figura 3. Tendencias en España.

Por último, se debe remarcar que Twitter permite realizar otro tipo de acciones como puede ser seguir a usuarios para que sus tweets aparezcan en tu página principal, o enviar mensajes privados a otros usuarios sin limitación de caracteres, permitiendo tener conversaciones privadas sin límites.

3.2. Métodos de interacción automática con Twitter

En este apartado se explicarán las principales formas de interactuar automáticamente con la plataforma Twitter, haciendo una pequeña comparativa entre ellas.

3.2.1. API de desarrolladores

Twitter pone a disposición de los desarrolladores el acceso a su API oficial para automatizar tareas. Para obtener acceso a esta API, el desarrollador deberá crear una cuenta en Twitter y solicitar los permisos para acceder a esta API, justificando las razones y objetivos de su uso. Los empleados de Twitter entonces valorarán si conceder el acceso o no al desarrollador solicitante.

Esta API de desarrolladores es, por lo tanto, una forma de acceder de forma autenticada a Twitter, por lo que todas las acciones que se realicen a través de la API se estarán haciendo a través de la cuenta de usuario asociada a la cuenta de desarrollador. Esto es, si se le da "me gusta" a un tweet, la cuenta que estará dando me gusta será la cuenta asociada.

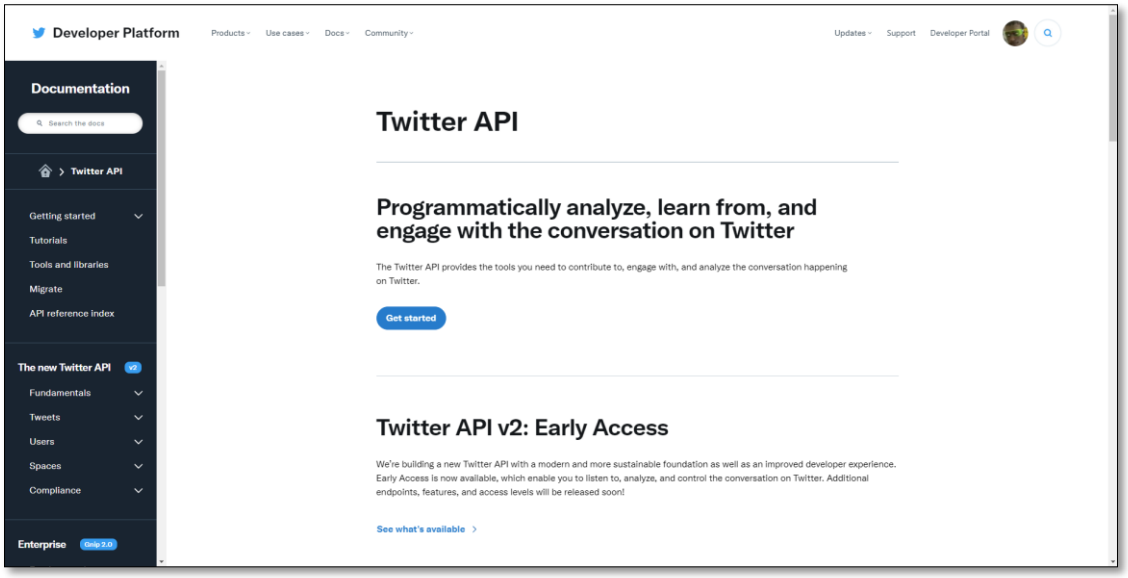

Figura 4. Dashboard del panel de desarrolladores.

El panel de desarrolladores permite crear proyectos y aplicaciones, generando una serie de claves secretas y tokens que serán utilizadas posteriormente en las llamadas a la API para autenticar a la aplicación.

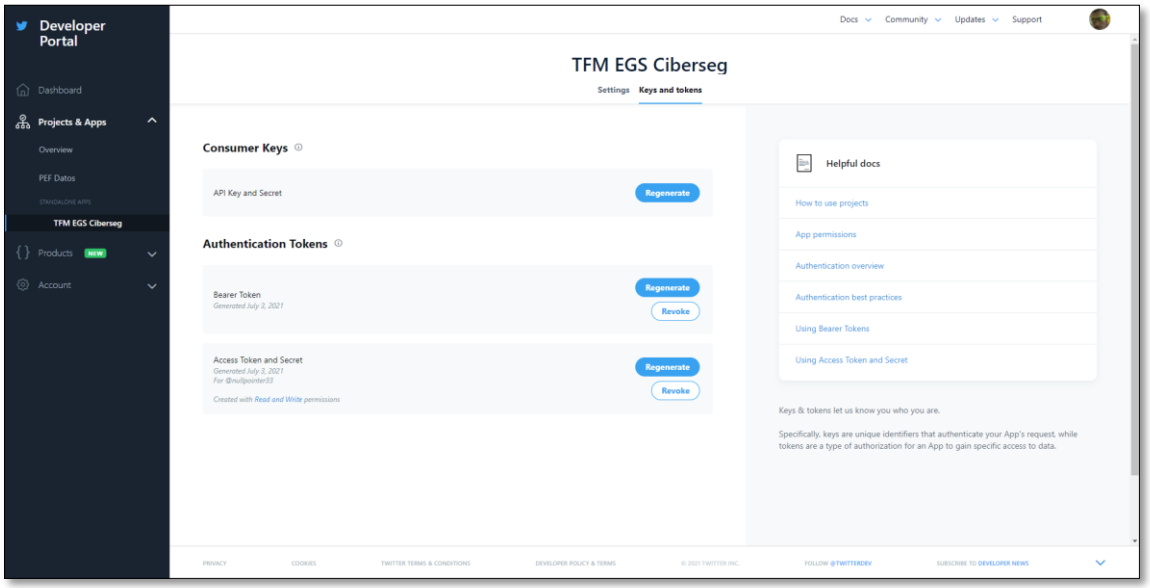

Figura 5. Aplicación creada para el TFM.

La API está implementada como una API ReST, por lo que funciona a través de HTTP. En la documentación oficial de Twitter, se pueden ver las llamadas que se deben lanzar contra la API para realizar según qué acciones.

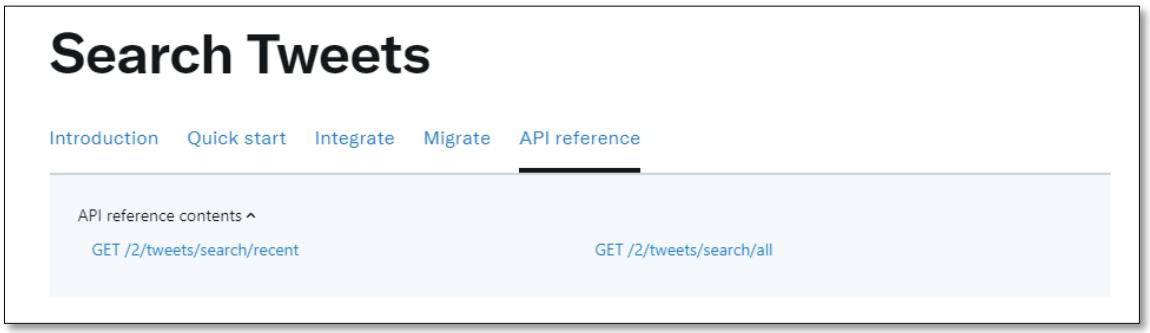

Figura 6. Ejemplo de llamadas a la API ReST de Twitter.

Para autenticar a la aplicación en la plataforma, Twitter permite tres formas de autenticación [9]:

- **Basic authentication**: haciendo uso de email y contraseña. Solo algunos *endpoints* muy específicos de la API premium permiten acceder mediante este método de autenticación.
- **OAuth 1.0a**: haciendo uso de claves secretas de la API y tokens de autenticación OAuth.
- **OAuth 2.0**: haciendo uso de *bearer tokens*. Este tipo de autenticación se hace para funciones muy específicas de la API que solo requieren tener permiso de lectura, ya que no autentica a la cuenta, simplemente permite realizar ciertas acciones.

Debido a que gestionar las llamadas a la API directamente requeriría escribir mucho código cada vez que se quiere implementar cualquier aplicación, por lo que han proliferado las librerías no

oficiales encargadas de hacer de intermediario entre el código de la aplicación y la API ReST de Twitter. Entre estas librerías se pueden contar Tweepy para Python3 o Twitter4J para Java.

La propia plataforma de Twitter proporciona algunas herramientas oficiales para interactuar con la plataforma en varios lenguajes como Python, JavaScript o Ruby, pero su funcionalidad es bastante más reducida que la de las librerías no oficiales.

En cuanto a funcionalidad, la API de Twitter permite realizar una gran cantidad de funciones, como las siguientes:

- Realizar búsquedas en Twitter bajo parámetros de búsqueda específicos. Solo puede encontrar tweets publicados hasta 7 días atrás desde el momento de la búsqueda, no muestra tweets más antiguos.
- Búsqueda de información sobre un tweet en concreto sin restricciones, en base a su ID.
- Recuperar información sobre cuentas de usuario:
    - Fecha de creación.
    - Tweets de la cuenta.
    - Seguidores y cuentas seguidas.
- Publicación de Tweets; desde tweets normales hasta responder a otros tweets, tweets citados, etc.
- Gestión de los "me gusta" de la cuenta. Permite dar, quitar y revisar los tweets a los que se ha dado *like*.
- Gestión de los retweets de la cuenta. Permite dar, quitar y revisar los tweets a los que se ha dado retweet.

Además, Twitter tiene una funcionalidad especial única de la API de desarrolladores que es el *endpoint* de *stream* de Twitter. Esto es, permite recuperar tweets publicados por los usuarios en tiempo real en base a un filtro definido por el desarrollador.

Por último, la API de desarrolladores de Twitter cuenta con una API premium, que permite acceder a los tres siguientes *endpoints*:

- **30-day search**: permite realizar búsquedas parametrizables de la misma manera que la API estándar de Twitter, pero expandiendo la búsqueda a aquellos tweets que se han publicado con una antigüedad de hasta 30 días.
- **Full archive search**: permite realizar búsquedas parametrizables de la misma forma que la API estándar de Twitter, expandiendo la búsqueda hasta el inicio de Twitter. Esto es,

permite recuperar cualquier tweet publicado en Twitter desde la creación de la plataforma (siempre que no haya sido borrado).

- **Account activity**: permite registrar un número de cuentas para realizar un seguimiento completo de su actividad en la red social, recuperando información como Tweets, tweets borrados, menciones, respuestas, *likes*, retweets, etc.

Es importante remarcar que todos los *endpoints* de la API de desarrolladores están limitados a un número de peticiones lanzadas por segundo, o tweets recuperados en un periodo de tiempo, incluyendo la API de stream, aunque estas restricciones son bastante relajadas y es difícil llegar a los límites establecidos.

En caso de las consultas premium, se tiene una versión básica que permite hacer un número muy reducido de consultas a esta API, pudiendo incrementar el límite establecido a través de una suscripción de pago mensual a la plataforma.

3.2.2. Métodos de scraping

La segunda opción de cara a interactuar con la plataforma es a través de métodos de *scraping* web.

Esto es, se podrán recuperar tweets de la plataforma a través de herramientas de *crawling*, *spidering* y *scraping* web, realizando búsquedas a través del propio buscador que Twitter pone a disposición de todos sus usuarios.

Existen múltiples soluciones de código abierto para afrontar este problema, como puede ser implementar un *scraper* con el framework **scrapy** [10], o utilizar soluciones especialmente diseñadas para el *scraping* de Twitter como **twint** [11]. La tercera solución sería implementar un *scraper* web enfocado en Twitter a través de librerías como **Selenium** [12].

En este proyecto se hará un estudio de los dos últimos métodos propuestos, esto es, twint y Selenium.

Twint es una herramienta de código abierto escrita en Python3 que tiene dos formas de operación:

- Como herramienta independiente a través de línea de comandos.

- Como librería de Python3, con el objetivo de permitir a los desarrolladores implementar su funcionalidad en sus aplicaciones.

Este *scraper* permite recuperar tweets parametrizando las consultas para ajustarse lo máximo posible a las necesidades del usuario, pero no permite autenticarse en la plataforma de ninguna forma. Por ello, a diferencia de las soluciones basadas en la API de desarrolladores, solo se podrán realizar acciones de "lectura" en la plataforma, esto es, no se podrán publicar tweets, dar "me gusta" o retweet a ningún tweet, etc.

Twint es una herramienta muy completa y avanzada de cara a realizar consultas en la plataforma, permitiendo lanzar consultas con los siguientes parámetros de configuración:

- Por nombre de usuario.
- Por palabras clave que deba contener un tweet.
- Por una fecha máxima y/o mínima.
- Anteriores a un año en concreto.
- Tweets que contengan contenido especial como números de teléfono, direcciones de correo electrónico.
- Tweets que provengan de cuentas verificadas en la plataforma.

El listado anterior no es exhaustivo en cuanto a la funcionalidad de customización de búsquedas de Twint, pero sirve para dar una visión general de lo que la herramienta permite.

Además, la plataforma permite recuperar otro tipo de datos como son los seguidores y cuentas seguidas de una determinada cuenta, así como los tweets a los que ha dado "me gusta", etc.

De cara a la exportación de datos, twint permite guardar los resultados de las consultas de las siguientes formas:

- Exportando a formato CSV.
- Exportando a un fichero de texto.
- Exportando a un fichero en formato JSON.
- Almacenando los resultados en una base de datos SQLite.
- Cargando los datos en un clúster de ElasticSearch directamente.

Como ventajas de hacer uso de twint frente a crear una aplicación que haga uso de la API oficial de desarrolladores se podría reseñar que no hay límites de consultas de ningún tipo, y que permite realizar consultas anónimas en la plataforma debido a que no se necesita iniciar sesión con ninguna cuenta para realizar las consultas.

En cuanto a las desventajas, tenemos la otra cara de la moneda, debido a que twint no permite autenticarse en la plataforma, tampoco permite ninguna de las funcionalidades de "escritura" mencionadas anteriormente. Además, debido a limitaciones en la paginación de tweets por parte de Twitter, en caso de que una consulta recupere más de 3200 resultados, twint solo podrá acceder a los primeros 3200, perdiendo el resto.

Además, durante las pruebas que se hicieron con la herramienta durante su estudio, se debe remarcar que los resultados de sus consultas son increíblemente inconsistentes. En ocasiones, lanzando varias veces la misma consulta se recuperaban resultados totalmente diferentes; en una ocasión en concreto se lanzó una consulta simple dos veces seguidas, recuperando un total de cero resultados la primera vez, y más de mil la segunda.

Figura 7. Logo de Selenium WebDriver.

Pasando al estudio de la segunda opción, Selenium es un framework enfocado a la realización de tests automatizados en aplicaciones web. Este framework cuenta con los siguientes componentes principales:

- **Selenium IDE**. Es un entorno integrado de desarrollo implementado a través de un *add-on* del navegador Firefox o una extensión de Chrome. Permite grabar, editar y realizar pruebas de depuración sobre aplicaciones web. Estos tests se guardan en un lenguaje de scripting especial llamado Selenese.

- **Selenium client API**. Para evitar que las pruebas deban ser escritas en el lenguaje Selenese, Selenium desarrolló una API en varios lenguajes de programación como Python, Java y C#, con el objetivo de permitir a los desarrolladores escribir sus programas en otros lenguajes y utilizar Selenium directamente desde ellos.

- **Selenium Webdriver**. Es un driver encargado de recoger los comandos escritos en Selenese, o desde la API de cliente, y enviárselos a un navegador web para que realice las acciones descritas en el código, obteniendo una serie de resultados.

A modo de resumen, Selenium permite escribir aplicaciones en lenguajes de alto nivel como Python o Java, con el objetivo de automatizar tareas en aplicaciones y servicios web a través de un driver de un navegador web como Google Chrome o Mozilla Firefox, emulando lo que haría un usuario normal de la aplicación.

Para el caso que nos compete, Selenium permite crear una solución que realice búsquedas automatizadas en la plataforma Twitter, permitiendo la autenticación con una cuenta de usuario, automatizando las tareas de “escritura” para las que twint no daba soporte.

Como contrapunto a esto, se debe remarcar que la plataforma Twitter actualizó sus términos de uso en 2017 [13] para prohibir ciertas acciones como las siguientes:

- Autenticarse en la plataforma mediante métodos que no sean la API oficial de desarrolladores.
- Enviar tweets o mensajes directos de forma automatizada si caen bajo la definición de SPAM, esto es:
    - Publicar sobre tendencias del momento en Twitter.
    - Publicar posts duplicados o substancialmente similares de forma repetida.
- Dar "me gusta" de forma automatizada a tweets.
- Seguir o dejar de seguir cuentas de usuarios, así como añadirlos a las colecciones.

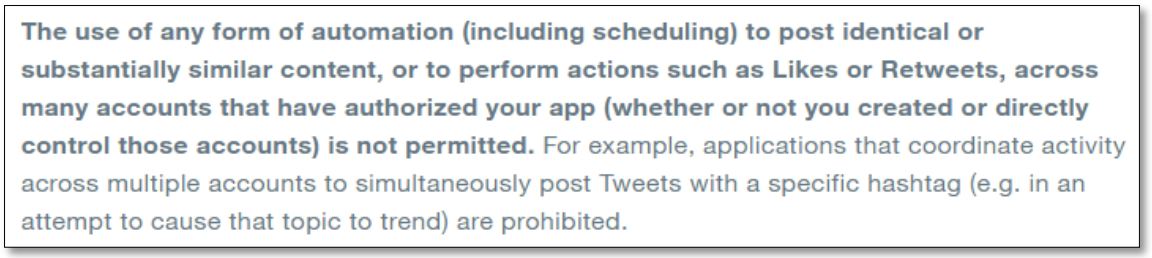

Figura 8. Comunicado de Twitter sobre la automatización de acciones. [14]

Esto es, aunque se pueda desarrollar una aplicación con todas las características de una creada haciendo uso de la API oficial de desarrolladores, esquivando de esta forma todos los límites de consultas a la API, las acciones realizadas con las cuentas asociadas a esta aplicación estarán incumpliendo los términos de uso y, en caso de que se detecte este tipo de actividad, serán suspendidas permanentemente de la plataforma.

3.3. Redes de bots en Twitter

En este apartado se hará un estudio de las redes de bots en Twitter, su funcionamiento, objetivos, y prevalencia en la plataforma.

Según un estudio publicado en 2017 [15], se estima que el número de cuentas en Twitter gestionadas totalmente por bots está entre el 9% y el 15% de las cuentas totales de la plataforma.

Sumado a estos sorprendentes números, se debe añadir la facilidad que hay actualmente para conseguir código capaz de realizar funcionalidades básicas de bot en esta red social. Sin ir más lejos, simplemente lanzando una consulta rápida en GitHub, con los términos de búsqueda "Twitter Bot", se recuperan más de 20000 repositorios públicos, de los cuales más de 9000 están escritos usando Python como lenguaje principal.

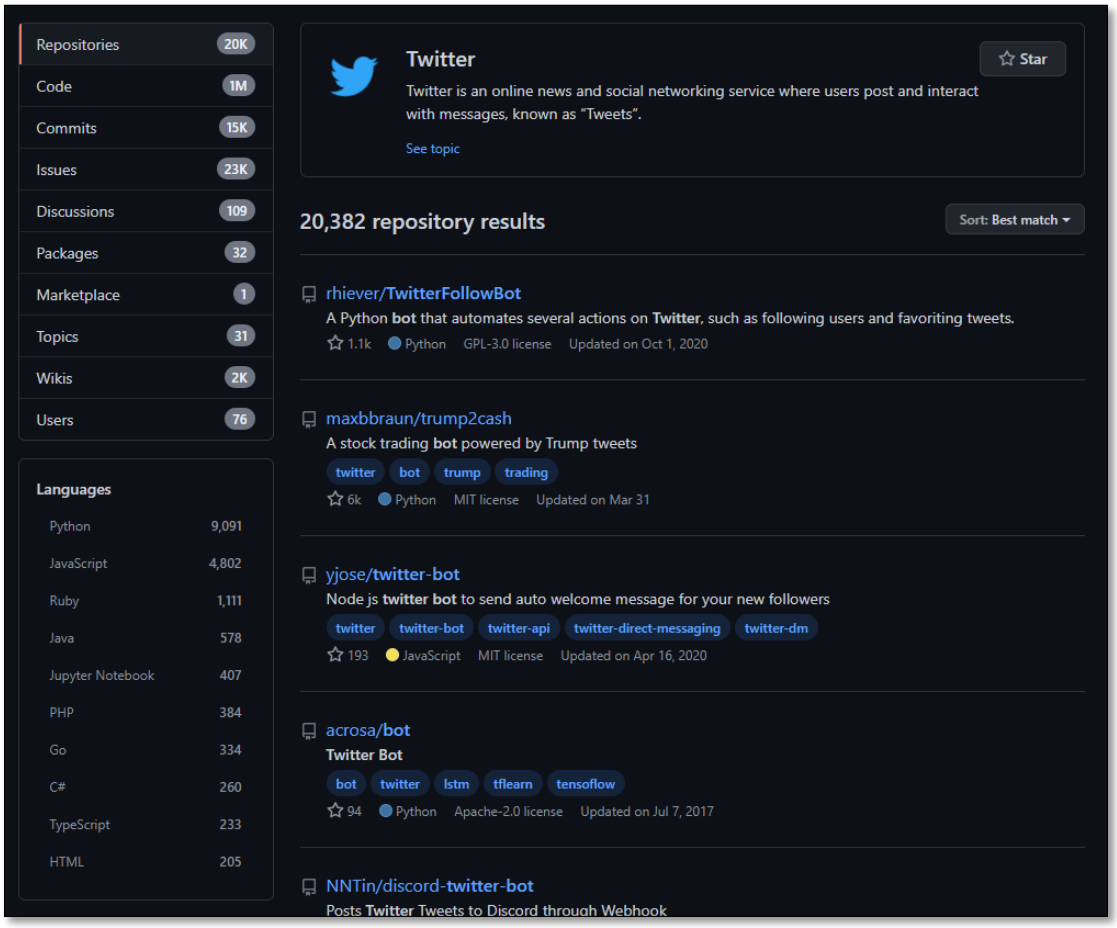

Figura 9. Resultado de búsqueda en GitHub sobre bots de Twitter.

Las redes de bots en Twitter pueden estar diseñadas con distintos objetivos en mente, pero principalmente se pueden categorizar de la siguiente manera:

- Campañas de SPAM para publicitar ciertos servicios o productos. Los bots normalmente publican tweets con enlaces que pasan por acortadores de URLs con anuncios para conseguir una remuneración.
- Campañas de phishing, tanto dirigidas como no dirigidas.
- Redes con la intencionalidad de publicitar información falsa para influir en la opinión popular en temas políticos.

En cualquier de estos casos, el objetivo principal de estos bots es conseguir el mayor público posible, por lo que las principales estrategias seguidas por estos bots se suelen centrar en las siguientes acciones:

- Engañar a los usuarios normales a seguir a estas cuentas para que sus publicaciones aparezcan en su *timeline* (pantalla de inicio).

- Envío de mensajes directos a usuarios.
- Y, principalmente, la adhesión a hashtags que se encuentren en tendencias en el momento de publicación de los tweets.

Aunque haya mucha información sobre cómo crear un bot que funcione sobre la plataforma Twitter, así como distintos métodos para detectar bots y redes de bots, realmente hay sorprendentemente poca información sobre el funcionamiento interno de redes de bots reales, esto es, no se ha encontrado ningún ejemplo de análisis de código, ni código por sí mismo, de una red de bots real. Si volvemos a realizar la búsqueda anterior en Github, podemos ver como el número de resultados se reduce a 24.

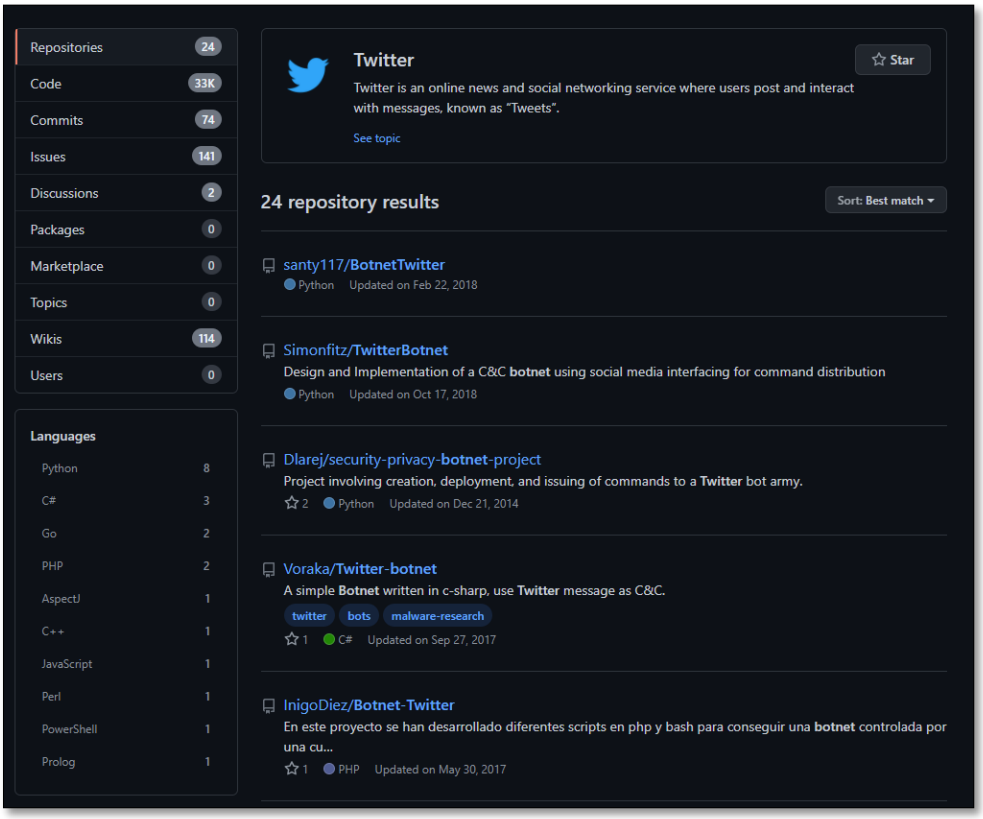

Figura 10. Resultado de búsqueda en GitHub sobre botnets de Twitter.

Durante el estudio que se ha realizado sobre el código de varias supuestas redes de bots para Twitter publicado en GitHub, la gran mayoría de estas redes hacían uso de la API de desarrolladores oficial de Twitter para autenticar sus cuentas, lo cual, en un entorno real, es totalmente impracticable, ya que, para acceder a la API de Twitter, la cuenta debe ser aprobada manualmente en el programa de desarrolladores por un empleado de la compañía.

Además, en ningún caso estas pruebas de concepto estudiadas se construían siguiendo una arquitectura distribuida. En todos los casos estudiados, el código de la botnet recogía varios bots, cada uno con una cuenta de desarrollador operativa, que trabajaban secuencialmente realizando determinadas acciones en función de lo que la red quería realizar.

Debido a esta falta de información sobre el funcionamiento de redes de bots reales en la plataforma, se ha decido hacer una implementación desde cero de esta, simulando una arquitectura de botnet maliciosa real, esto es, siguiendo una arquitectura cliente-servidor entre los bots y un servidor de Command & Control (C2).

4. Desarrollo del sistema

Este apartado trata la arquitectura de los sistemas desarrollados, entrando en detalle en los distintos componentes que conforman cada sistema software, haciendo hincapié en las partes más importantes y complejas de cada uno de estos componentes.

4.1. Aplicación para búsquedas en Twitter a través de su API

En este primer subapartado, se tratará todo lo referente a la aplicación de escritorio desarrollada con el fin de darle una interfaz de acceso a los usuarios a Twitter a través de llamadas a la API oficial de desarrolladores.

4.1.1. Arquitectura de la aplicación

Esta primera aplicación cuenta con las siguientes partes:

- **Aplicación de escritorio:** componente principal del sistema. Los usuarios serán capaces de realizar búsquedas en Twitter que podrán ser visualizadas a través de la interfaz gráfica de usuario de la aplicación, serán capaces también de guardar listas de usuarios para futuras búsquedas, y de exportar los resultados de estas búsquedas a formato CSV.

- **Interfaz de acceso a Twitter mediante la API:** componente lógico que se encuentra dentro del código fuente de la aplicación de escritorio. Este componente es el encargado de comunicarse con la API de Twitter, haciendo las consultas que requiera el usuario a través de la interfaz.

- **Base de datos:** pequeña base de datos cuyo objetivo es almacenar las listas de usuarios guardados para poder realizar futuras búsquedas.

A continuación, se presenta un diagrama que resume las interacciones entre las distintas partes de esta aplicación:

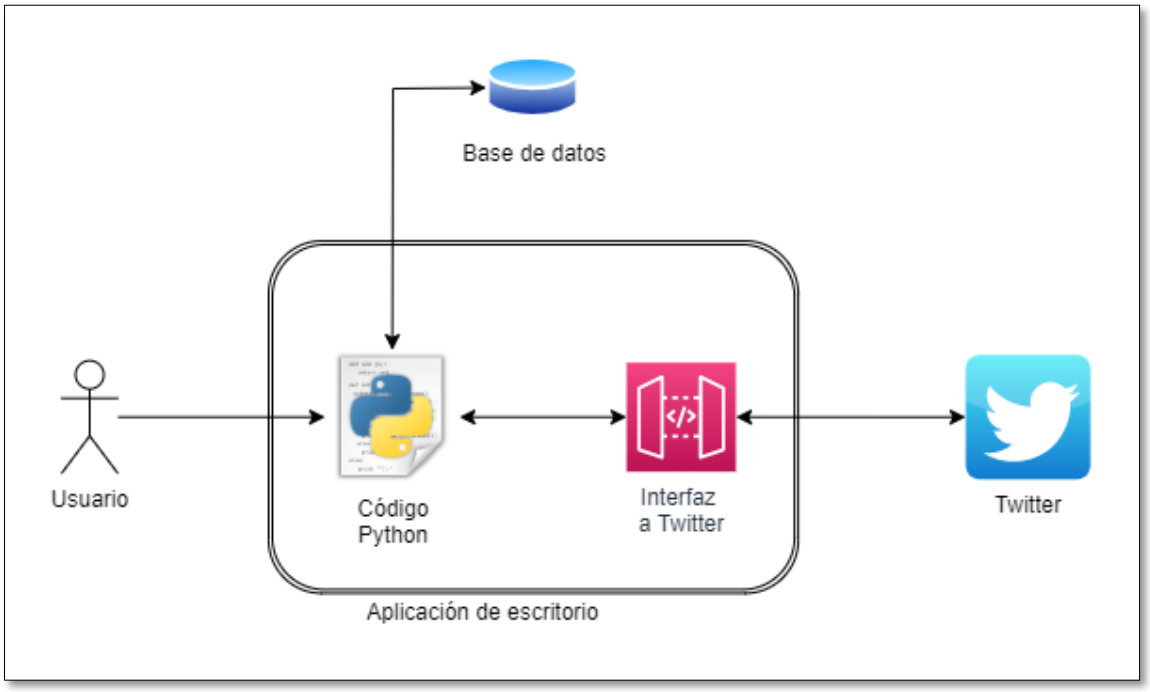

Figura 11. Arquitectura general del sistema.

Para el correcto desarrollo de la aplicación propuesta se necesitan los siguientes componentes:

- Ordenador personal con Python3 instalado con los módulos necesarios para el funcionamiento de la aplicación (especificados en el archivo *requirements.txt* del repositorio de GitHub).
- Base de datos SQLite3 instalada en el mismo ordenador personal.
- Cuenta de desarrollador de Twitter (o acceso a una a través de las claves secretas de la API), para poder hacer las llamadas a la API oficial de desarrolladores.

4.1.2. Base de datos

Para gestionar los datos referentes al almacenamiento de cuentas para su seguimiento de forma *offline*, es decir, sin tener la necesidad de seguirlas en Twitter, se ha creado el siguiente modelo da base de datos:

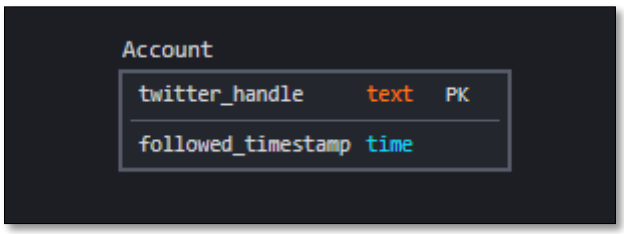

Figura 12. Diagrama de la base de datos.

La tabla representada en la figura anterior tiene la siguiente estructura:

- **Tabla Account:** la tabla recoge la información referente a los datos de la cuenta que se está siguiendo, con el objetivo de poder realizar búsquedas rápidamente en la API de Twitter limitando los resultados a las cuentas a las que se esté siguiendo. Sus columnas son las siguientes:

    - **Twitter_handle:** Tipo text, primary key de la table. Almacena el *handle* (nomber de la cuenta) de Twitter del usuario al que estamos siguiendo.

    - **Followed_timestamp:** Tipo time. Almacena el timestamp del momento en el que se añadió al usuario a la lista de usuarios seguidos.

Debido a la simplicidad del modelo de datos de la aplicación, se ha decidido hacer uso de **SQLite 3**, ya que permite la creación de bases de datos ligeras y muy fáciles de implementar y mantener cuando la base de datos tiene estas características. Ya que SQLite no cuenta con un gestor de conexiones remotas, la aplicación de escritorio y la base de datos deben estar en la misma máquina física para su correcto funcionamiento.

Figura 13. Logo de SQLite.

SQLite no cuenta con algunos de los tipos de datos especificados anteriormente. Por ejemplo, el tipo boolean se almacena como un integer con valores [0, 1], los tipos time se almacenan como text, etc. A efectos prácticos que no existan estos tipos no es relevante ya que se pueden hacer gestiones externas bastante sencillas para suplir esta falta de tipos.

Con la finalidad de que se pueda probar el funcionamiento de la aplicación, se ha creado un script SQL encargado de instanciar la base de datos. Este script se puede encontrar en el directorio de la base de datos del repositorio de GitHub.

4.1.3. Aplicación de escritorio

La aplicación de escritorio es el eje central de este sistema, y también la parte que cuenta con una mayor complejidad.

La idea principal de la que parte el desarrollo es darle al usuario la posibilidad de realizar búsquedas en Twitter a través de una interfaz de usuario simple e intuitiva, con la posibilidad de crear una serie de listas de seguimiento de usuarios para personalizar sus búsquedas, y con la

posibilidad de exportar los resultados del uso de esta aplicación de alguna forma con el objetivo de continuar con su análisis en aplicaciones de terceros.

Debido al resultado del estudio de las diferentes soluciones de cara a construir la interfaz de comunicación con Twitter, se decidió implementar esta aplicación en el lenguaje de programación **Python3**. Además, Python3 tiene una gran interoperabilidad con SQLite, pudiendo simplificar lo máximo posible las llamadas a la base de datos.

Figura 14. Logo de Python.

De cara a la construcción de la interfaz gráfica de usuario, se barajaron las librerías más populares para construir este tipo de interfaces de usuario. Entre ellas, se barajó la posibilidad de hacer uso de Tkinter [16] o Kivy [17], pero se acabó escogiendo la implementación para Python del **framework Qt** [18].

Qt es una librería multiplataforma que permite desarrollar interfaces gráficas de una forma rápida, sencilla y obteniendo un resultado limpio y fácil de utilizar y entender por los usuarios finales.

El framework Qt trae consigo una aplicación de escritorio llamada **Qt Designer** [19], que permite diseñar las interfaces de forma muy intuitiva, y guardando el resultado de este diseño en un formato propio, que más adelante, mediante otras herramientas del framework, permite generar código en distintos lenguajes de programación que se encargue de generar estas interfaces tal y como las habíamos creado. Esta herramienta ha sido utilizada durante el desarrollo para crear las interfaces gráficas de esta aplicación, exportando las interfaces y generando el código correspondiente en Python3.

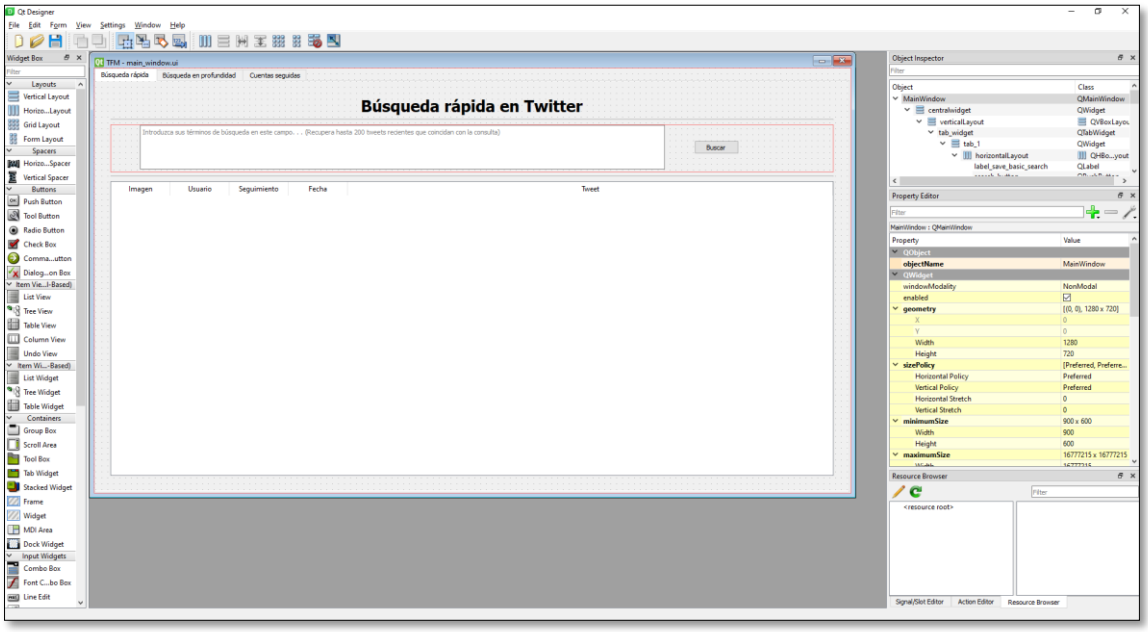

Figura 15. Interfaz principal de la aplicación abierta en Qt Designer.

En cuanto al mecanismo para recuperar datos de la plataforma Twitter, se barajaron dos acercamientos distintos en base a lo estudiado en el estado del arte:

1. Realizar las consultas a través de la API oficial de desarrolladores de Twitter.

2. Realizar las consultas a través de alguna herramienta de terceros de *scraping* de la red social.

En concreto, para Python, se consideraron las librerías **Tweepy** de cara a hacer uso de la API oficial de Twitter, y **Twint** de cara a hacer uso de un sistema de *scraping* que no necesitase autenticarse en la plataforma.

Después de hacer un estudio de las diferentes opciones que se tenían y de las ventajas y desventajas de cada una, se decidió implementar la aplicación haciendo uso de la API oficial de desarrolladores de Twitter haciendo uso de la librería Tweepy, debido principalmente a que se tenía una cuenta aceptada por la plataforma de antemano, y que el uso de esta API, aunque las consultas estén limitadas por la plataforma, trae muchas más opciones y posibilidades que haciendo uso de soluciones de *scraping*.

La aplicación permite realizar las siguientes acciones:

- **Búsquedas rápidas en Twitter**: Este tipo de búsqueda recupera los 200 Tweets más recientes que coincidan con el criterio de búsqueda introducido por el usuario. No tiene límites de uso, más allá de los límites de consultas/segundo que establece Twitter.

- **Búsquedas con la API premium de Twitter**: La API premium de Twitter permite hacer 2 tipos de búsqueda distintas, ambas limitadas a un número determinado de consultas. Estas son:

    - **API de Archivo**: permite recuperar todos los Tweets archivados por Twitter, esto es, desde el primer tweet publicado el 21 de marzo de 2006.

    - **API 30-day**: permite hacer búsquedas de Tweets publicados en los últimos 30 días en la plataforma.

- **Seguimiento de usuarios**: La aplicación permite seguir a aquellos usuarios que consideremos interesantes para, posteriormente, poder realizar consultas específicas que busquen tweets solo de esos usuarios, con el objetivo de hacer un seguimiento de su actividad.

- **Exportación de los resultados de búsqueda**: Se permite al usuario exportar los resultados de cualquiera de las búsquedas (rápida, archivo, 30-day) a formato CSV, con el objetivo de que pueda ser cargado en otra aplicación para su posterior análisis.

La estructura de ficheros de esta aplicación es la siguiente:

- **main.py:** fichero Python que recoge el main de la aplicación, se encarga de instanciar los *handlers* de Twitter y la base de datos, así como la interfaz gráfica de usuario.

- **main_window.py:** fichero Python que recoge toda la funcionalidad de la aplicación, así como el código responsable de generar y gestionar la interfaz gráfica de usuario.

- **database_handler.py:** fichero Python que se encarga de gestionar la conexión con la base de datos, así como de lanzar las consultas necesarias.

- **twitter_handler.py:** fichero Python encargado de gestionar la conexión con la API de Twitter, así como de lanzar las consultas necesarias. Será tratado en un apartado separado, debido a su importancia.

Empezando por el fichero **main.py**, tenemos un fichero de código sencillo, cuya totalidad de código son las siguientes 14 líneas:

```
import main_window
from database_handler import DatabaseHandler
from twitter_handler import TwitterHandler


def main():
    twitter_handler = TwitterHandler()
    database_handler = DatabaseHandler()
```

```
    main_window.Ui_MainWindow(twitter_handler, database_handler)
if __name__ == "__main__":
    main()
```

Como podemos ver, este fichero de código se encarga de ejecutar la función *main( )* cuando se cumpla la condición de que este fichero es el que ha sido pasado como parámetro al intérprete de Python3 para su ejecución, es decir, que sea el fichero principal.

La función *main( )* se encarga de instanciar los gestores de la base de datos y Twitter, y la interfaz gráfica, pasándole como parámetro al constructor estos gestores para su uso interno. Una vez se ha instanciado la ventana principal de la aplicación, esta clase pasará a ser el "nuevo *main*" de facto de la aplicación, siendo la clase Ui_MainWindow el nexo de unión de todas las partes de la aplicación.

Pasando al fichero de código **database_handler.py**, tenemos una clase Python encargado de realizar toda la interacción necesaria por la aplicación principal con la base de datos SQLite.

La forma de actuación de todos los métodos de esta clase es la misma, y sigue el siguiente orden lógico:

1. Se abre la conexión con la clase de datos SQLite, almacenada localmente.
2. Se crea un cursor en la base de datos sobre el que ejecutar la consulta.
3. Se ejecuta la consulta correspondiente.
4. Se analizan los resultados de la consulta y:
    a. Se notifica al usuario en caso de error.
    b. Si hay que devolver el resultado de la consulta a la capa de aplicación, se hace *return* de este resultado.
5. Se cierran el cursor y la conexión con la base de datos para liberar recursos del sistema.

A continuación, se muestran ejemplos de código de cada uno de los distintos tipos de consulta que hay implementados en esta clase de gestión de la base de datos, que son:

- Inserción de datos.
- Consulta de datos.
- Eliminación de datos.

En cuanto a la inserción de datos, el siguiente método recoge el código encargado de añadir una nueva cuenta de usuario a la base de datos de usuarios seguidos, una vez el usuario pulsa el botón correspondiente en la interfaz gráfica de usuario:

```
def add_account(self, twitter_handle):
    try:
        con = sqlite3.connect("./src/resources/database/DB.db")
        cursor = con.cursor()

        cursor.execute(
            f"INSERT INTO Account VALUES('{twitter_handle.strip()}',
             '{self.__get_timestamp()}');"
        )
        con.commit()

        cursor.close()
        con.close()

    except:
        print("Error while attempting to create a new followed account.")
```

Como podemos ver, el método se encarga de insertar en la base de datos el *handler* de la cuenta de usuario a la que el usuario ha decidido seguir, entrando en el método como un parámetro que le deberá ser pasado desde la clase que gestione la interfaz de usuario.

El segundo valor que se introduce en la tabla *Account* es un timestamp para registrar el momento en el que se hizo este seguimiento, y podemos ver que este valor lo estamos recuperando directamente desde el método *__get_timestamp( )* de la propia clase *DatabaseHandler*.

Este método es un método auxiliar encargado de generar un timestamp en el formato que permite a SQLite3 operar sobre el directamente, ya que, como hemos dicho anteriormente, SQLite3 no soporta directamente el tipo de datos time/date, y es almacenado como text. El tipo text no tiene restricciones de ningún tipo sobre cómo debe ser almacenado, pero si queremos utilizar las funciones de tiempo que SQLite aporta, debemos almacenar los timestamps en un formato concreto.

En nuestro caso concreto no sería necesario, ya que no hacemos uso de estas funciones internas de SQLite, pero por motivos de escalabilidad y calidad de código se ha decidido seguir este formato.

El código de esta función auxiliar es el siguiente:

```
def __get_timestamp(self):
    return datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

Como podemos ver, devuelve un string con la fecha en formato “YYYY-MM-DD hh:mm:ss”.

De cara a la consulta de datos, el siguiente método se encarga de recuperar todos los datos que tenga la base de datos con respecto a las cuentas de usuario seguidas. Este método es llamado cuando el usuario pulsa el botón de actualizar en la ventana de la interfaz encargada de mostrar las cuentas que tiene seguidas.

```
def read_followed_accounts(self):
    con = sqlite3.connect("./src/resources/database/DB.db")
    cursor = con.cursor()

    cursor.execute("SELECT * FROM Account;")

    result = cursor.fetchall()
    cursor.close()
    con.close()

    return result
```

Este fragmento de código lanza la consulta contra la base de datos y devuelve el resultado a la capa de aplicación en bruto, es decir, directamente una vez se ha recuperado de la base de datos, para su posterior tratamiento y adecuación al formato en el que será presentado al usuario.

Por último, se muestra un ejemplo de eliminación de tuplas de la base de datos.

```
def delete_followed_account(self, twitter_handle):
    con = sqlite3.connect("./src/resources/database/DB.db")
    cursor = con.cursor()

    cursor.execute(
        f"DELETE FROM Account WHERE twitter_handle='{twitter_handle.strip()}';"
    )
    con.commit()

    rows = cursor.rowcount

    cursor.close()
    con.close()

    if rows != 1:
        print("Error while trying to delete followed twitter account.")
```

Como podemos ver, el método se encarga de eliminar de la base de datos el registro cuya *primary key* es el *handler* de la cuenta de usuario almacenada en la base de datos. Este *handler* es pasado como un parámetro desde la clase que gestiona la interfaz de usuario.

Para comprobar que el borrado se ha hecho correctamente, podemos hacer uso del atributo *rowcount* del cursor que ha ejecutado la consulta. Al realizar borrados de la base de datos, el cursor almacena el número total de registros que se han borrado de la base de datos, por lo que, si este número es distinto de 1, sabemos que ha habido algún tipo de error durante el borrado.

Continuamos con la explicación del fichero que recoge toda la lógica de la aplicación, así como el código de gestión de la interfaz gráfica, **main_window.py**.

Este archivo recoge un total de 3 clases de Python, dos de ellas son clases auxiliares que sirven para extender la usabilidad de la aplicación a través de mecanismos que requieren hacer uso de la herencia. Las clases son las siguientes:

- **ImageLoader:** clase que hereda de Qtcore.Qobject. Creada para gestionar la carga de ciertos elementos en la interfaz de usuario en paralelo a la ejecución del hilo principal, para evitar que sea bloqueante.

- **ClickableLabel:** clase que hereda de QtWidgets.Qlabel. Esta clase se ha tenido que implementar para suplir la necesidad de crear *labels* en la interfaz gráfica que pudiesen ser clickadas como si fuesen botones, ya que, por defecto, este tipo de elementos no permite registrar eventos de click.

- **Ui_MainWindow:** clase principal de la aplicación, recoge todo el código tanto de la interfaz gráfica como la lógica encargada de gestionar las distintas partes de la aplicación. Es el nexo de unión de todas las partes de este sistema, encargándose también de comunicarse con el gestor de Twitter y el gestor de la base de datos.

Una vez conocemos todas las clases de esta aplicación, se presenta el siguiente diagrama para dar una visión global de las interacciones de estas clases durante la ejecución:

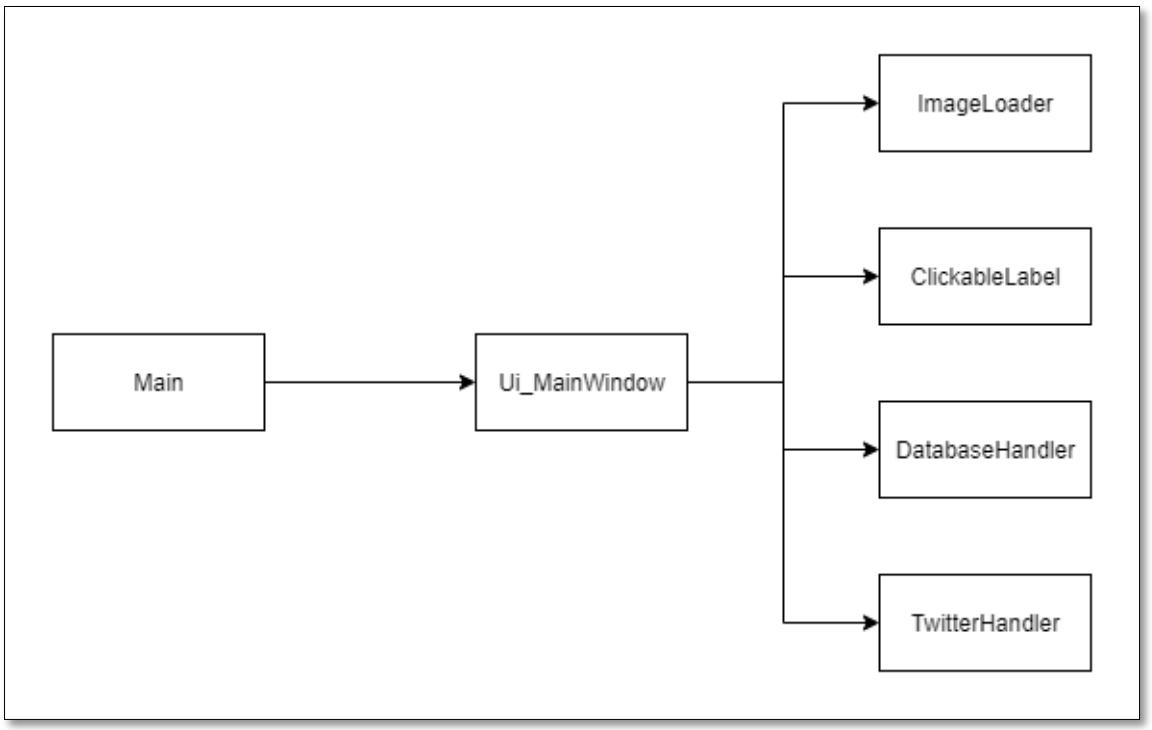

Figura 16. Diagrama de la arquitectura de las clases de la aplicación de escritorio.

Debido a que este archivo de código cuenta con poco menos de 1000 líneas de código, se entrará en detalle solo en las partes que se consideren más relevantes para su documentación en esta memoria.

Empezaremos analizando el constructor de la clase:

```
def __init__(
        self, twitter_handler: TwitterHandler, database_handler: DatabaseHandler
    ):
        super().__init__()
        self.twitter_handler = twitter_handler
        self.database_handler = database_handler

        self.last_basic_tweets = []
        self.last_advanced_tweets = []
        self.following_list = []

        app = QtWidgets.QApplication(sys.argv)
        app.setWindowIcon(QtGui.QIcon("./src/resources/images/logo_master.jpg"))
        MainWindow = QtWidgets.QMainWindow()
        self.setupUi(MainWindow)

        MainWindow.show()
        sys.exit(app.exec_())
```

Este constructor se encarga de asignar los *handlers* de Twitter y la base de datos que recibe como parámetro como atributos de la clase. Después, crea unas listas que serán utilizadas más adelante para realizar las exportaciones a formato CSV como atributos de la clase.

Una vez ha hecho esto, instancia la interfaz gráfica haciendo las llamadas a los métodos correspondientes de la librería PyQt (implementación en Python del framework Qt), y acaba enlazando la señal de salida de la ventana principal con la terminación del programa, esto es, que cuando el usuario cierre la ventana principal, el programa termine con su ejecución.

El método llamado en el constructor, *setupUi(MainWindow)*, es uno de los dos métodos autogenerados por Qt Designer. Este método se encarga de crear todos los elementos de la interfaz, darles forma, configurar sus dimensiones, colocación, y prepararlos para que funcionen en tiempo de ejecución.

Este método ha sido modificado para enlazar ciertas funcionalidades como puede ser que cuando se haga click en un botón se enlace con la función correspondiente. Este tipo de eventos de click, se hace mediante *signals* en el framework Qt.

```
self.search_button.clicked.connect(self.basic_twitter_search)
self.advanced_search_button.clicked.connect(self.advanced_twitter_search)
self.update_follower_list_button.clicked.connect(self.update_following_list)
self.radioButton_full_archive.clicked.connect(self.radio_full_archive_clicked)
self.radioButton_30_day.clicked.connect(self.radio_30_day_clicked)
```

Aquí tenemos un ejemplo de cómo se hace este registro. Para simplificar, lo que se está declarando en este fragmento de código es que cuando cierto botón es pulsado, se ejecute la función que se

la pasa por parámetro, es decir, se conecta la señal de pulsar el botón con la función correspondiente.

El segundo método autogenerado por Qt Designer se llama *retranslateUi(MainWindow)*, y se encarga simplemente de rellenar todos aquellos componentes que tienen texto con sus correspondientes strings. Este método es creado de esta forma para que sea más fácil cargar unos u otros strings si se quiere dar soporte multilenguaje para la aplicación; en nuestro caso, solo tiene strings de soporte para español.

Una vez se han explicado los componentes necesarios para instanciar y lanzar la interfaz, se procede a explicar la funcionalidad de la aplicación. Para ello, primero debemos ver cómo está organizada la interfaz gráfica a través de la siguiente figura con un ejemplo de un resultado al lanzar una búsqueda rápida:

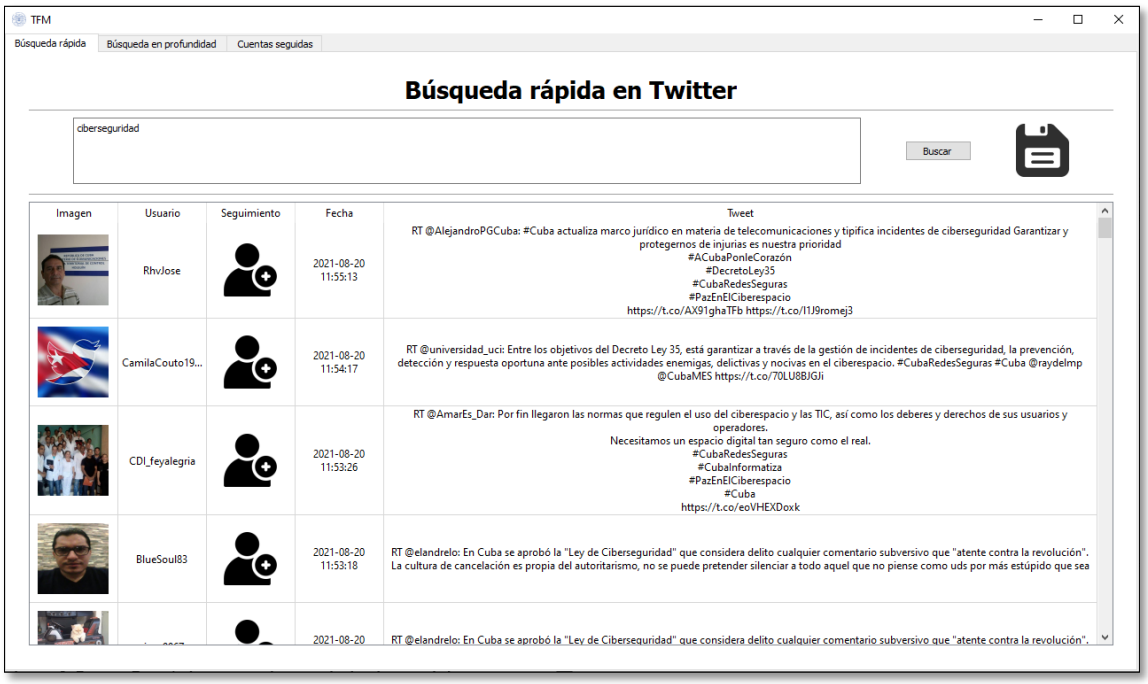

Figura 17. Búsqueda rápida en Twitter a través de la aplicación de escritorio.

Como podemos ver, la aplicación está compuesta por una serie de pestañas en las que tenemos unos menús que nos permiten realizar distintos tipos de búsquedas en Twitter. En el caso del ejemplo, tenemos el resultado de haber realizado una búsqueda con el término “ciberseguridad”, haciendo uso del *endpoint* de búsqueda rápida de la API de Twitter.

Cada una de estas pestañas tiene la misma estructura, que es la siguiente:

1. Etiqueta con el título y objetivo de la pestaña.

2. Cuadro de mando donde el usuario puede añadir parámetros de búsqueda a la consulta que se lanzará a Twitter, así como guardar los resultados de la consulta si así lo desea.

3. Tabla en la que se muestran los resultados de la búsqueda.

Para cada una de las distintas búsquedas, existe un método encargado de hacerle la petición al *handler* de Twitter, y rellenar la tabla con los resultados que ha obtenido. Para ello, de forma general, se siguen los siguientes pasos.

En primera instancia, se recoge el input del usuario desde la interfaz, y si hay algún tipo de error como, por ejemplo, que no haya rellenado ningún campo de búsqueda, se lanza un mensaje de error para advertírselo y se cancela la consulta:

```
query = self.textEdit_advanced_query.toPlainText()
handle = self.textEdit_user_handle.toPlainText()

if query.strip() == "" and handle.strip() == "":
    QtWidgets.QMessageBox().critical(
        self.centralwidget,
        "Error",
        "Debe introducir al menos una consulta o una cuenta de usuario para realizar
la búsqueda",
    )
```

Y el resultado en interfaz sería el siguiente:

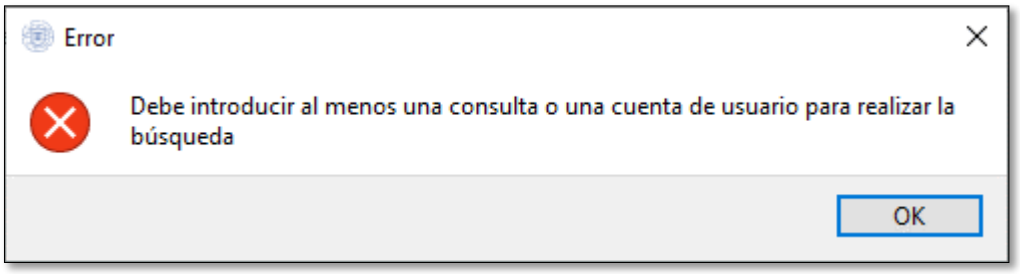

Figura 18. Error al lanzar consulta en la aplicación de escritorio.

En caso contrario, si el input del usuario es correcto, se crea un menú de carga bloqueante para que el usuario sepa que la aplicación está haciendo la consulta y no pueda seguir interactuando con la aplicación. Para crear estos menús de carga, se ha extraído la funcionalidad a un método auxiliar para que fuese reutilizable en distintas zonas de la aplicación. El código responsable es el siguiente:

```
def __loading_menu(self, string, horizontal_size=250):
    dialog = QtWidgets.QDialog(self.centralwidget)
    dialog.setWindowTitle(f"Cargando {string}. . .")
    dialog.resize(horizontal_size, 1)
    dialog.open()

    return dialog
```

Y el resultado en la interfaz sería la siguiente barrita bloqueante, que desaparece cuando la búsqueda ha completado:

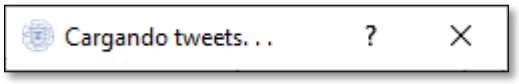

Figura 19. Mensaje de carga de tweets.

Una vez la interfaz de usuario queda bloqueada para que el usuario no pueda tocar nada, el programa seguirá corriendo de fondo resolviendo la consulta y cargando los resultados en la tabla. Para ello, lo primero que hace es eliminar todo el contenido que pudiese tener la tabla anteriormente poniendo el contador de filas a 0, después lanza una consulta que devuelve *N* resultados y genera *N* filas nuevas que llenará con estos resultados. También guardará este resultado en un atributo de la clase por si acaso el usuario desease exportarlo a CSV.

```
loading_dialog = self.__loading_menu("tweets", horizontal_size=250)
profile_image_list = []
self.tableWidget_simple.setRowCount(0)
self.last_basic_tweets = self.twitter_handler.custom_twitter_search(query)
self.tableWidget_simple.setRowCount(len(self.last_basic_tweets))
```

Una vez se han creado las columnas, es necesario crear cada una de las celdas y rellenarlas con el contenido. Por ello, para cada uno de los resultados se toman los siguientes pasos:

```
for i in range(len(self.last_basic_tweets)):
    item = QtWidgets.QTableWidgetItem()
    item.setTextAlignment(QtCore.Qt.AlignHCenter | QtCore.Qt.AlignVCenter)
    self.tableWidget_simple.setItem(i, 0, item)
    item = QtWidgets.QTableWidgetItem()
    item.setTextAlignment(QtCore.Qt.AlignHCenter | QtCore.Qt.AlignVCenter)
    self.tableWidget_simple.setItem(i, 1, item)
    item = QtWidgets.QTableWidgetItem()
    item.setTextAlignment(QtCore.Qt.AlignHCenter | QtCore.Qt.AlignVCenter)
    self.tableWidget_simple.setItem(i, 2, item)
    item = QtWidgets.QTableWidgetItem()
    item.setTextAlignment(QtCore.Qt.AlignHCenter | QtCore.Qt.AlignVCenter)
    self.tableWidget_simple.setItem(i, 3, item)
    item = QtWidgets.QTableWidgetItem()
    item.setTextAlignment(QtCore.Qt.AlignHCenter | QtCore.Qt.AlignVCenter)
    self.tableWidget_simple.setItem(i, 4, item)

    profile_image_list.append(
        self.last_basic_tweets[i].user.profile_image_url_https
    )

    followed_image_label = self.__get_followed_image_label(
        self.last_basic_tweets[i].user.screen_name
    )

    self.tableWidget_simple.item(i, 1).setText(
        self.last_basic_tweets[i].user.screen_name
```

```
    )
    self.tableWidget_simple.setCellWidget(i, 2, followed_image_label)
    self.tableWidget_simple.item(i, 3).setText(
        self.last_basic_tweets[i].created_at.strftime("%Y-%m-%d %H:%M:%S")
    )
    self.last_basic_tweets[i].full_text = (
        f"RT @{self.last_basic_tweets[i].retweeted_status.user.screen_name}: "
        + self.last_basic_tweets[i].retweeted_status.full_text
        if self.last_basic_tweets[i].full_text.startswith("RT @")
        else self.last_basic_tweets[i].full_text
    )
    self.tableWidget_simple.item(i, 4).setText(
        self.last_basic_tweets[i].full_text
    )
```

Esto es:

1. Se inicializan las celdas de la fila para que puedan ser rellenadas más adelante.
2. Se añade la URL de la imagen de perfil del usuario a una lista para su posterior tratamiento.
3. Se recupera la imagen de estado de seguimiento en un *label* creado con la clase descrita al inicio, para que sea clickable como un botón.
4. Se carga toda la información en las celdas de la tabla.

Se han mencionado los *labels* clickables, esta es la primera de las dos clases auxiliares que se han creado en este fichero de código, con el objetivo de incrementar la funcionalidad de los *labels* de Qt habituales. El código de la clase es el siguiente:

```
class ClickableLabel(QtWidgets.QLabel):
    clicked = QtCore.pyqtSignal()

    def mouseReleaseEvent(self, QMouseEvent):
        if QMouseEvent.button() == QtCore.Qt.LeftButton:
            self.clicked.emit()
```

Hereda de QLabel y simplemente implementa el tratamiento de la señal de clickado cuando un usuario hace click en este tipo de *labels*, posibilitando enlazar este clickado con la llamada a una función; en el caso de la tabla, se crean imágenes clickables para permitir al usuario añadir a los usuarios a las listas de seguimiento.

Una vez se ha rellenado toda la tabla, solo queda cargar las imágenes de perfil. Para ello, se lanzará un hilo con el objetivo de que se haga en paralelo, evitando que la interfaz de usuario quede bloqueada durante mucho tiempo, debido a que el procesado de las imágenes es lo que más tarda

en todo el proceso de búsqueda y presentación de los datos. A parte de esto, se deberá cerrar el mensaje que le indicaba al usuario que la aplicación estaba buscando resultados.

El código responsable de esto es el siguiente:

```
self.launch_profile_image_thread(
    self.tableWidget_simple, profile_image_list
)
loading_dialog.close()
```

Y el código encargado de lanzar el hilo es el siguiente:

```
def launch_profile_image_thread(self, table, profile_image_list):
    self.thread = QtCore.QThread(self.centralwidget)
    self.worker = ImageLoader()
    self.worker.moveToThread(self.thread)

    self.thread.started.connect(
        lambda: self.worker.run(self, table, profile_image_list)
    )
    self.worker.progress.connect(self.load_image)
    self.worker.finished.connect(self.thread.quit)
    self.worker.finished.connect(self.worker.deleteLater)
    self.thread.finished.connect(self.thread.deleteLater)

    self.thread.start()
```

El hilo es una de las dos clases auxiliares mencionadas anteriormente, y su código es el siguiente:

```
class ImageLoader(QtCore.QObject):
    finished = QtCore.pyqtSignal()
    progress = QtCore.pyqtSignal(QtWidgets.QTableWidget, int, QtWidgets.QWidget)

    def run(self, ui_window, table, image_list):
        for i in range(len(image_list)):
            QtWidgets.QApplication.processEvents()
            self.progress.emit(
                table, i, ui_window.get_profile_image_label(image_list[i])
            )
            QtWidgets.QApplication.processEvents()
        self.finished.emit()
```

La clase hereda de los objetos QObject del framework Qt, y funciona de la siguiente manera:

1. Se registran las señales *finished* y *progress* para poder comunicarse con el hilo principal de la aplicación.

2. Cuando se lanza el hilo, se ejecuta la función *run*, que, para cada imagen se encarga de:

    a. Fuerza a la interfaz a que se actualice.

b. Procesa la imagen de perfil.

c. Envía una señal al hilo de la interfaz para que este sepa que está preparada y pueda cargarla.

d. Fuerza a la interfaz a que se actualice.

Hay varios puntos interesantes en los que profundizar en estas líneas de código, y se deben a limitaciones del framework Qt con el tema de trabajar en paralelo.

Para empezar, el único hilo al que se le permite modificar la interfaz de cualquier forma es el hilo principal, cualquier otro hilo, incluyendo en esta descripción el hilo que tenemos cargando imágenes, no podría cargas las imágenes directamente, sino que se debe limitar a procesar la imagen y pasársela al hilo principal para que este las cargue en sus respectivas celdas.

Además, cuando la interfaz gráfica está siendo actualizada de forma tan rápida, queda en un estado en el que se bloquea esperando a que se realicen todos los cambios antes de actualizarse, con el objetivo de no hacer una sobrecarga de procesamiento debido al refresco de la interfaz varias veces por segundo.

Por ello, si la interfaz no es forzada a refrescarse antes y después de cargar la imagen, esta queda bloqueada de la misma forma que si el hilo principal fuese el encargado de procesar las imágenes, haciendo que procesarlas en paralelo fuese totalmente inútil.

Para solventar esto, se hace la llamada a *QtWidgets.QApplication.processEvents( )* antes y después de cargar la imagen, forzando al hilo principal a actualizar la interfaz gráfica.

El código que se ha mostrado para demostrar el funcionamiento de la carga de datos en las tablas de la aplicación ha sido el correspondiente al de la pestaña de búsquedas sencillas. Cada uno de estos métodos encargados de rellenar las tablas tiene sus peculiaridades para adaptarse a las características de la tabla, pero de forma general todas funcionan tal y como se ha visto anteriormente.

Alguna de estas particulares puede ser en el caso de la búsqueda en profundidad el parseo de las distintas opciones que se le da al usuario, o, en el caso de la carga de las cuentas seguidas en la aplicación, cargar los datos desde la base de datos SQLite3 en vez de hacer la llamada a Twitter.

A continuación, se presentan imágenes de las ventanas de búsqueda en profundidad y seguimiento de usuarios:

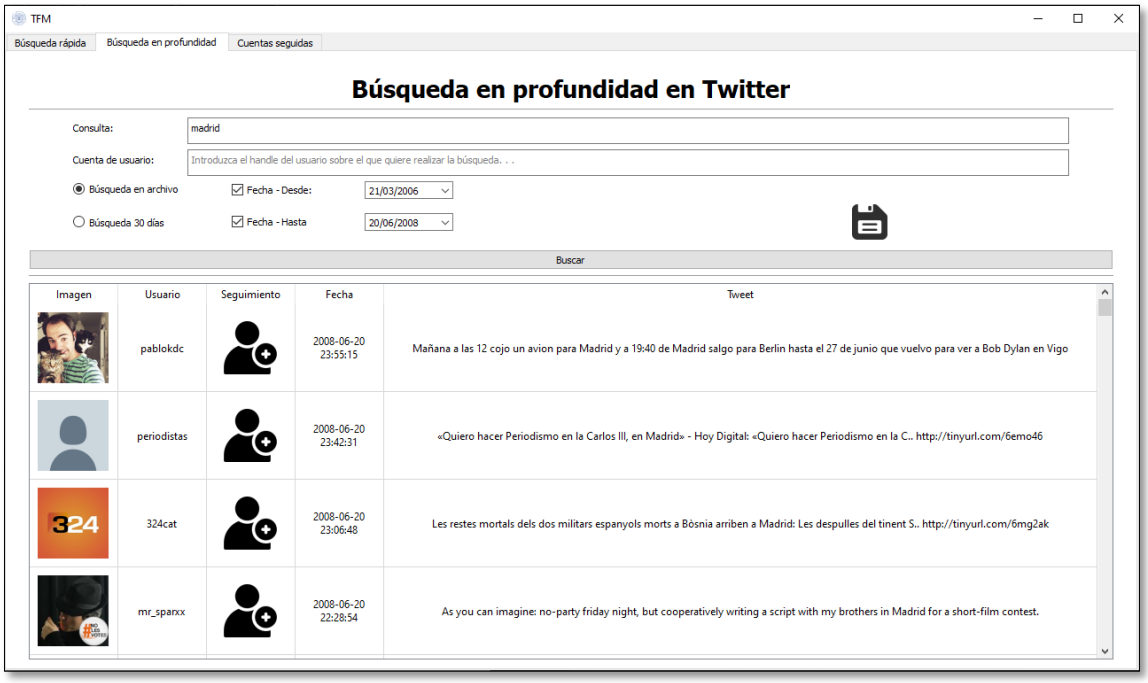

Figura 20. Ventana de búsqueda en profundidad en la aplicación de escritorio.

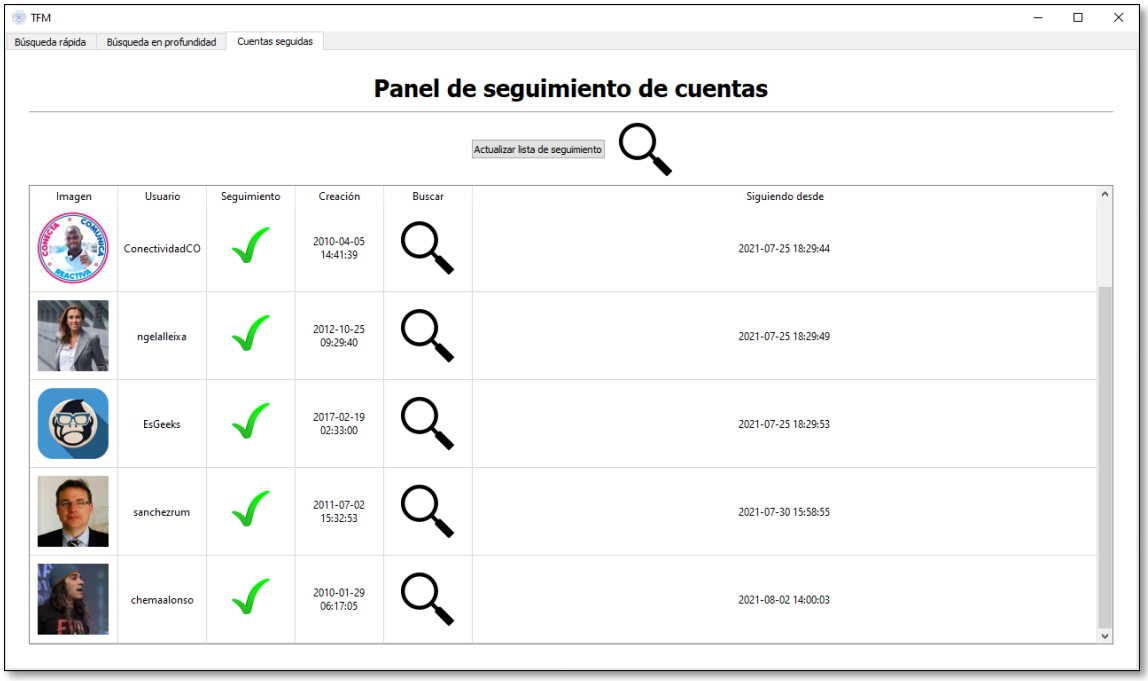

Figura 21. Panel de seguimiento de cuentas en la aplicación de escritorio.

Con esto concluye la explicación de cómo se muestran los resultados de las consultas lanzadas por los usuarios en la interfaz de usuario.

En cuanto a la exportación de datos, se puede realizar tanto en búsquedas simples como en cualquiera de los dos tipos de búsqueda en profundidad, a través de los botones de guardado que se pueden observar en la siguiente figura:

Figura 22. Botón de guardado en la interfaz de usuario.

La aplicación permite exportar todos los datos a formato CSV, a través de dos funciones separadas, pero esencialmente similares, una para las búsquedas rápidas y otra para las búsquedas en profundidad. El siguiente fragmento de código corresponde a la exportación de una búsqueda rápida:

```
def save_basic_search(self):
    if not self.last_basic_tweets:
        self.__show_save_error()
    else:
        message_box = self.__generate_message_box()
        message_box.setText(
            "¿Desea guardar la última consulta rápida en formato CSV?"
        )
        result = message_box.exec_()

        if result == QtWidgets.QMessageBox.Yes:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            with open(
                f"./src/saved_queries/simple_search_result_{timestamp}.csv", "w"
            ) as f:
                writer = csv.writer(f)
                writer.writerow(["id", "screen_name", "created_at", "text"])
                for tweet in self.last_basic_tweets:
                    writer.writerow(
                        [
                            tweet.id,
                            tweet.user.screen_name,
```

```
                    tweet.created_at,
                    tweet.full_text.encode("utf-8"),
                ]
            )
```

El código se encarga de recuperar los datos del atributo encargado de almacenar la lista con los resultados de la última búsqueda que ha hecho el usuario.

En caso de que esta lista esté vacía, es decir, que el usuario no haya hecho ninguna consulta, salta un error y se cancela la exportación. En caso contrario, si la lista de Tweets no está vacía, se pide confirmación al usuario y se exporta en formato CSV, almacenando todos estos resultados en un fichero marcado por el *timestamp* actual.

Por último, en cuanto a la funcionalidad de seguimiento, la aplicación permite mantener listas de seguimiento de las cuentas que el usuario considere interesantes. Al realizar una consulta, podemos ver el siguiente botón para cada resultado:

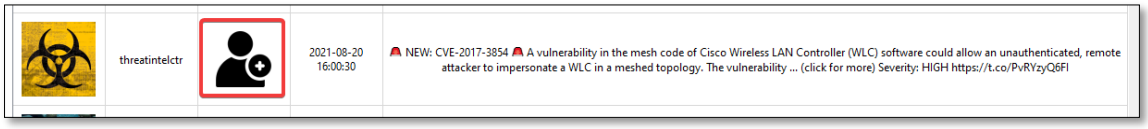

Figura 23. Botón de seguimiento en la interfaz de usuario.

Al pulsar ese botón, se le pide al usuario confirmar si realmente quiere seguir a esa cuenta de Twitter:

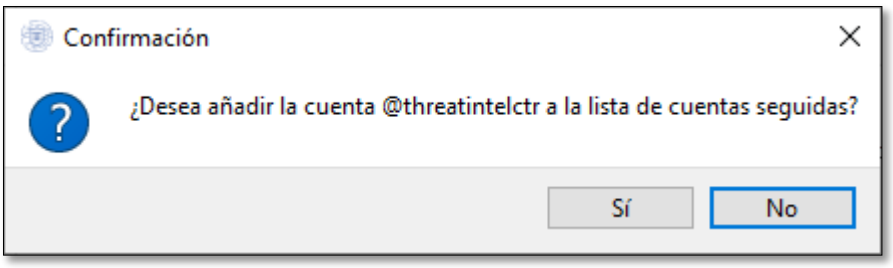

Figura 24. Mensaje de confirmación al seguir a una cuenta.

En caso de que acepte, esta cuenta se añadirá a la lista de cuentas seguidas, que puede ser consultada al hacer click en el botón de actualizar del panel de cuentas seguidas:

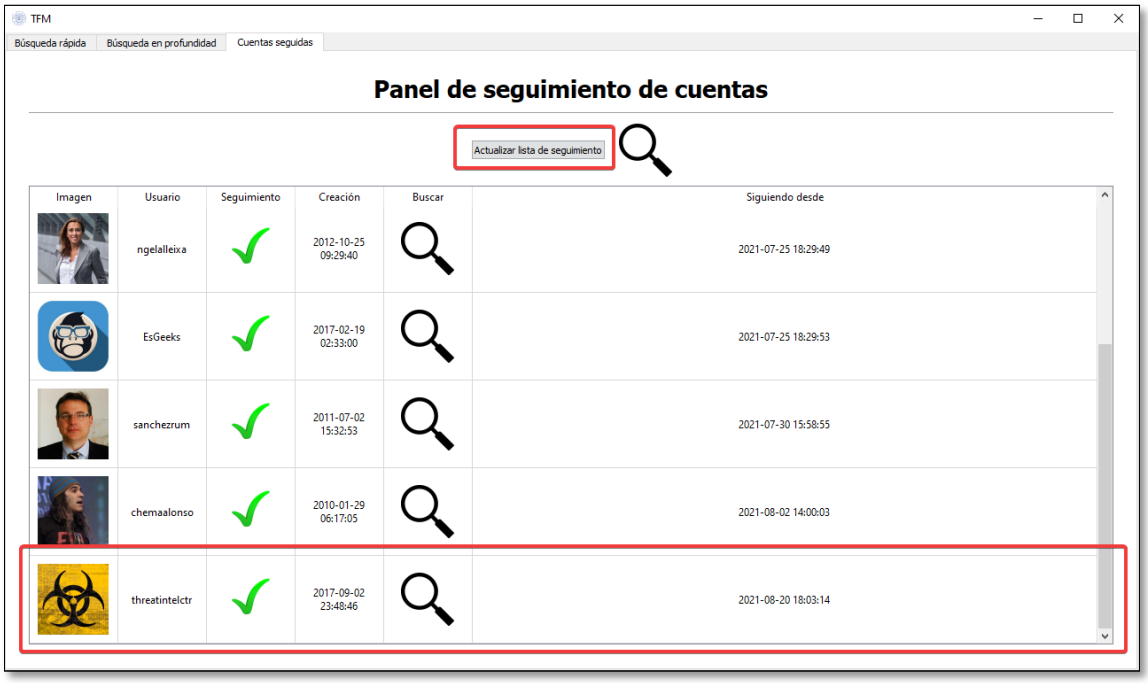

Figura 25. Botones para realizar búsquedas limitadas a las cuentas que se tienen seguidas.

Este panel tiene una segunda funcionalidad, que consiste en permitir lanzar consultas a la API de búsqueda rápida de Twitter, obteniendo solo resultados de la cuenta o cuentas a las que el usuario haya decidido seguir. El código encargado de gestionar estas búsquedas es el siguiente:

```
def lookup_following_tweets(self):
    if not self.following_list:
        QtWidgets.QMessageBox().critical(
        self.centralwidget,
        "Error",
        "Debe actualizar la lista de cuentas seguidas antes de hacer la consulta (y t
ener, al menos, una cuenta seguida).",
        )
    else:
        str_following = "("
        for account in self.following_list:
            str_following += f"from:{account[0]} OR "
        str_following = str_following [:len(str_following)-4] + ")"

        self.tab_widget.setCurrentIndex(0)
        self.textEdit.setText(str_following)
        self.search_button.click()

def lookup_following_tweets_single_account(self, handle):
        self.tab_widget.setCurrentIndex(0)
        self.textEdit.setText(f"from:{handle}")
        self.search_button.click()
```

La primera función se encarga de gestionar la búsqueda rápida de todas las cuentas que se encuentran en la lista de seguimiento del usuario, mientras que la segunda se encarga de gestionar la búsqueda de una sola cuenta de las que tiene seguidas. Como se puede ver, simplemente carga el texto de la consulta en el cuadro de texto del panel de búsqueda rápida y pulsa el botón para lanzar la consulta.

Si el usuario hace click en la lupa de la tabla, se lanzará una búsqueda rápida que recuperará los Tweets solo de ese usuario. Si, por el contrario, el usuario hace click en la lupa encontrada al lado del botón de actualización de la lista de seguimiento, se hará una búsqueda rápida que obtendrá resultados de todas las cuentas que se encuentren en la lista de seguimiento, como se puede observar en la siguiente figura:

Figura 26. Resultado de lanzar una búsqueda limitada a las cuentas seguidas.

Con esto concluye la documentación de la clase principal de la aplicación, así como de la interfaz de usuario. Obviamente no se ha hecho un análisis exhaustivo debido a las casi 1000 líneas de código de este fichero, pero se ha hecho hincapié en las partes más importantes para dar una visión general del funcionamiento de este sistema.

4.1.4. Interfaz de interacción con Twitter

La interfaz de interacción con Twitter es un componente lógico del sistema que realmente es una parte de la aplicación de escritorio, pero sobre la cual se hace esta distinción debido a la separación que se le ha dado en el código, y debido a la importancia que tiene para el funcionamiento del sistema.

Esta interfaz, es una clase escrita en Python3, que sirve para suplir las necesidades del usuario de cara a interactuar con la API de desarrolladores de Twitter. Como ya se ha anticipado anteriormente, esta clase hace uso de la librería **Tweepy** para hacer las llamadas a esta API de desarrolladores.

Al instanciar la clase, lo primero que se va a ejecutar es el código contenido en el constructor de esta, que es el siguiente:

```
def __init__(self):
    self.__initialize_api()
    self.__validate_login()
```

Lo primero que se va a ejecutar antes de nada es una inicialización de la API, autenticando al sistema con las claves secretas de la aplicación que hemos creado en el panel de desarrolladores de Twitter. El código que se encarga de esta inicialización y login es el siguiente:

```
def __initialize_api(self):
    self.__print_separator()
    print("Initializing Twitter bot. . .")

    with open("./src/resources/api_keys", "r") as keys:
        try:
            print("Attempting to read API keys file. . .")

            file_content = keys.readlines()
            consumer_key = file_content[0].replace("\n", "")
            consumer_secret = file_content[1].replace("\n", "")
            access_token_key = file_content[2].replace("\n", "")
            access_token_secret = file_content[3].replace("\n", "")

            auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
            auth.set_access_token(access_token_key, access_token_secret)

            self.api = tweepy.API(auth)

        except SystemExit:
            sys.exit(-1)
        except:
            print("Error while reading API keys from file, program will terminate")
            sys.exit(-1)
```

Como podemos observar, las claves de la API se recuperan de un fichero que se encuentra en la carpeta de recursos del sistema. El fichero deberá tener las claves correspondientes, una por cada línea, para que puedan ser leídos por la aplicación.

Una vez se ha leído el fichero completamente y se han recuperado las claves, la aplicación se autentica en Twitter a través de la API, y se guarda el objeto resultante de conectarse a la API en un atributo para que pueda ser utilizado en el resto de los métodos de la clase.

Lo segundo que se ejecuta al instanciar el *handler* de Twitter, es la verificación de que efectivamente el login ha sido correcto. En caso contrario, se lanza un mensaje de error y se para la ejecución, a la espera de que el usuario cree un fichero con claves secretas válidas con el que la aplicación pueda autenticarse en Twitter.

```
def __validate_login(self):
    print("Verifying credentials. . .")

    if self.api.verify_credentials():
        print("Authentication OK")
        self.__print_separator()
    else:
        print("Error during authentication, program will terminate")
        sys.exit(-1)
```

Una vez la aplicación ha validado que el login ha sido satisfactorio, se lanzará la interfaz gráfica y el usuario podrá hacer uso de la aplicación.

Cuando el usuario hace click en algún botón de lanzamiento de consultas a Twitter de la interfaz gráfica de usuario, se hace una llamada al método correspondiente de esta clase.

A continuación, se presenta el código encargado de gestionar una consulta rápida con el sistema:

```
def custom_twitter_search(self, search_param):
    return self.api.search(search_param.strip(), tweet_mode="extended", count=200)
```

Se lanza la consulta a Twitter con los parámetros que ha introducido el usuario y se recuperan los tweets en modo extended, esto es, recogiendo todo el texto del tweet en lugar de la versión recortada que la devuelve por defecto. Además, se especifica el parámetro *count* para limitar la búsqueda a 200 resultados.

En cuanto a los tweets recuperados por la API de Twitter, se puede remarcar que no se hace una distinción entre los tweets que haya publicado un usuario frente a los tweets a los que un usuario haya dado retweet, y que, para recuperar el texto completo del tweet en caso de que sea un retweet la forma de actuar es distinta a sí estamos recuperando un tweet normal, y por ello se pueden ver fragmentos de código en la clase main_window.py donde se analiza la respuesta de la API en busca de si el tweet es un retweet o no.

Esto se puede apreciar en el siguiente operador ternario a la hora de rellenar la celda correspondiente en la tabla de resultados de una búsqueda rápida.

```
self.last_basic_tweets[i].full_text = (
    f"RT @{self.last_basic_tweets[i].retweeted_status.user.screen_name}: "
    + self.last_basic_tweets[i].retweeted_status.full_text
    if self.last_basic_tweets[i].full_text.startswith("RT @")
    else self.last_basic_tweets[i].full_text
)
```

O en el siguiente método encargado de parsear los resultados de una búsqueda a la API premium:

```
def __recover_archive_tweet_text(self, tweet):
    if tweet.truncated:
        str_tweet = tweet.extended_tweet["full_text"]
    else:
        if hasattr(tweet, "retweeted_status"):
            if tweet.retweeted_status.truncated:
                str_tweet = f"RT @{tweet.retweeted_status.user.screen_name}: {tweet.r
etweeted_status.extended_tweet['full_text']}"
            else:
                str_tweet = f"RT @{tweet.retweeted_status.user.screen_name}: {tweet.r
etweeted_status.text}"
        else:
            str_tweet = tweet.text

    return str_tweet
```

Volviendo al código del *handler* de Twitter, tenemos dos métodos encargados de gestionar las llamadas a la API premium de 30 días y archivo de Twitter. Los dos métodos son prácticamente iguales, por lo que solo se va a mostrar y explicar el código de uno de ellos:

```
def thirty_day_search(self, query, handle="", fromDate="-1", toDate="-1"):
    full_query = query if handle == "" else f"{query} from:{handle}"

    if fromDate != -1:
        from_date = self.__format_date_to_archive(fromDate) + "0000"
        if toDate != -1:
            to_date = self.__format_date_to_archive(toDate) + "2359"
            result = self.api.search_30_day(
                "tfm30day", full_query, fromDate=from_date, toDate=to_date
            )
        else:
            result = self.api.search_30_day(
                "tfm30day", full_query, fromDate=from_date
            )
    else:
        if toDate != -1:
            to_date = self.__format_date_to_archive(toDate) + "2359"
            result = self.api.search_30_day("tfm30day", full_query, toDate=to_date)
        else:
            result = self.api.search_30_day("tfm30day", full_query)

    return result
```

El método anterior es el responsable de gestionar las llamadas al *endpoint* 30-day de la API de Twitter.

Como podemos ver, lo primero que va a hacer es recomponer la consulta a partir de los términos de búsqueda que haya introducido el usuario, y de la cuenta sobre la que quiere buscar en caso de que se haya especificado.

Después, se hace una evaluación de si el usuario ha establecido unos límites de fecha tanto inferior como superior a la hora de realizar la consulta, y se hace la llamada correspondiente a la API de Twitter a través de Tweepy en función de estos parámetros.

Para hacer llamadas a la API con fechas, es necesario parsear estas fechas a un formato concreto, para lo que se ha implementado una función auxiliar encargada de ello.

```
def __format_date_to_archive(self, date):
    formatted_date = date.split("/")
    str_formatted_date = formatted_date[2] + formatted_date[1] + formatted_date[0]

    return str_formatted_date
```

Con esto concluye la exploración de la clase responsable de interactuar con la API de desarrolladores de Twitter, y también el fragmento de la memoria destinado a explicar la aplicación de escritorio para realizar búsquedas en Twitter.

4.2. Botnet de Twitter

En este segundo subapartado, se tratará todo lo referente al sistema de red de bots o botnet desarrollado a través de herramientas de *scraping* web, así como todas las decisiones y problemas que se dieron durante este desarrollo.

La botnet está organizada de la siguiente manera:

- **Servidor Command & Control (C2)**: este servidor se encarga de monitorizar Twitter en busca de tweets que se consideren relevantes en base a un parámetro de configuración. Cuando recupera un tweet que cumpla con estas características, se lo envía a los bots en forma de comando.

- **Bots de la red**: los bots de la red se conectan inicialmente con el servidor C2, y se quedan a la espera de comandos. Cada vez que reciben un comando (tweet), le darán like, retweet, y comentarán en el tweet.

La botnet también se ha implementado haciendo uso del lenguaje **Python3**, debido principalmente a la facilidad que se tiene con respecto a las librerías para recuperar datos de Twitter, como ya se ha explicado anteriormente, y a las librerías que permiten la automatización de tareas en Twitter a través de medios no oficiales, como puede ser a través del *scraping* web.

De cara a la automatización de las acciones que los bots realizan sobre la plataforma, se ha decidido hacer uso de la librería **Selenium**, ya que permite automatizar estas acciones a través de un *webdriver* para un navegador web.

Es importante remarcar que el acceso y el uso de Twitter a través de herramientas automatizadas que no sean la API oficial de desarrolladores está prohibido y queda reflejado en los términos de usuario de la plataforma. Twitter cuenta con varias formas de detectar si el acceso a la plataforma se está haciendo de forma automatizada, y llega a desactivar las cuentas de las que detecta actividad inusual de este tipo.

De hecho, en el momento en el que se está escribiendo la memoria, una de las cuentas utilizadas para realizar pruebas sobre este sistema ha sido suspendida indefinidamente de la plataforma:

Figura 27. Trágico final de una de las cuentas utilizadas para probar la botnet.

De cara a recuperar los tweets de la plataforma en tiempo real, se ha decidido hacer uso de la API de desarrolladores de Twitter directamente, debido a que ya se tenía una cuenta y se creía que iba a ser lo más sencillo de implementar y desplegar para obtener una solución rápida y eficiente.

Para interactuar con esta API, se ha vuelto a hacer uso de la librería **Tweepy**, pero esta vez se ha implementado la funcionalidad de *stream* [20] de tweets que permite a los desarrolladores con acceso a la API recuperar tweets en tiempo real en función de unos parámetros de búsqueda predefinidos.

En el caso de una botnet real, los responsables de esta arquitectura muy probablemente no tengan una cuenta de desarrollador activa, o no quieran utilizarla debido a que sería mucho más sencillo detectar su implicación a través del comportamiento de esta cuenta.

Si se quisiese construir la misma arquitectura que se ha montado para este proyecto, esto también se podría hacer desarrollando código basado en la librería de Selenium, recuperando tweets constantemente en base a unos criterios de búsqueda específicos y filtrando en base a su antigüedad, para evitar que se enviasen tweets repetidos a la botnet, consiguiendo que el servidor estuviese prácticamente leyendo tweets en tiempo real.

Entrando más en detalle en la solución desplegada, el proyecto cuenta con la siguiente estructura de ficheros:

- **c2_bot_server.py**: fichero python que recoge la clase C2Server. Este fichero es el servidor de *Command & Control* de la botnet, recuperando tweets de la red social y enviándoselos como comandos a la red de bots.

- **bot_client.py**: fichero de código que recoge la clase BotClient. El fichero implementa toda la funcionalidad de los bots, y está preparado para funcionar tanto en red como en local, permitiendo incluso varias instancias por máquina.

- **bot[x].cfg**: fichero de configuración necesario para el correcto funcionamiento de los bots. Debe ser pasado como parámetro al bot al ejecutarlo. Este fichero de configuración almacena las credenciales de la cuenta de usuario que utilizará el bot, así como el nombre de la cuenta para evitar limitaciones de Twitter que serán explicadas más adelante.

Empezando por el funcionamiento del servidor C2, su flujo de ejecución es bastante sencillo y sigue el siguiente esquema:

1. El servidor se autentica el Twitter a través de las claves secretas de la API de desarrolladores de Twitter.

2. Se lanza un hilo encargado de escuchar en un puerto a conexiones UDP entrantes para registrar a bots que se vayan conectando a la lista de bots en activo.

3. Se lanza un segundo hilo encargado de hacer uso de la API de *stream* de Twitter, recuperando tweets en tiempo real en base a una consulta preestablecida.

4. Cuando un usuario publica un tweet que coincida con la consulta, se encapsula en un comando y se envía a los bots a través de UDP.

En cuanto a los bots de la red, seguirán el siguiente flujo de ejecución:

1. El bot cliente recupera las credenciales de la cuenta de usuario de Twitter que utilizará.

2. Inicializa el *webdriver* de Chrome para Selenium.

3. Se autentica en la red social con las credenciales.

4. Manda un mensaje UDP al servidor para registrarse en la lista de bots activos.

5. Al recibir confirmación del servidor, se queda a la espera de recibir comandos.

6. Cuando recibe un comando (tweet), accede a través del *webdriver* a la URL del Tweet, le da Like, Retweet, y comenta un mensaje preestablecido.

4.2.1. Código del servidor

Pasamos a explorar el código responsable de hacer funcionar al servidor. Para empezar, se debe remarcar que para poder implementar la API de *stream* en Twitter con Tweepy, es necesario crear una clase que herede de *tweepy.StreamListener*, debido a que se debe implementar al menos el método *on_status*, que se verá más adelante.

El código de la clase empieza con el siguiente constructor, que hace las veces de main del servidor. Desde él, se inicializan todos los atributos e hilos necesarios para el correcto funcionamiento de esta infraestructura. El código es el siguiente:

```
def __init__(self):
    self.bot_list = []
    self.__initialize_api()
    self.__validate_login()

    Thread(target = self.register_bot).start()

    self.tweet_stream = tweepy.Stream(self.api.auth, self)
    self.tweet_stream.filter(track=['egs_tfm_ciberseg'], is_async=True)
```

Lo primero que hace es inicializar la lista de bots registrados con una lista vacía, para poder ir añadiéndolos más adelante según se vayan registrando.

Lo segundo que hará es inicializar la conexión con la API intercambiando las claves secretas y, seguidamente, validar que la autenticación ha sido correcta. Este código es exactamente igual que el explicado en el apartado de la aplicación de escritorio, por lo que no se entrará más en detalle sobre él.

La tercera acción del constructor es inicializar el hilo encargado de registrar bots al recibir conexiones.

Por último, lanza el hilo encargado de escuchar el *stream* de tweets de la plataforma. Se lanza en un hilo aparte gracias al parámetro *is_async* inicializado a True. Tal y como está configurado en la captura del código, el servidor recogerá todos los tweets que contengan de alguna forma la cadena de caracteres “egs_tfm_ciberseg”. Esta ha sido la cadena utilizada durante las pruebas para contener las acciones de la botnet, evitando que se “boteasen” tweets reales.

Lo siguiente que se debe explicar, es el código que ejecuta el hilo encargado de registrar los bots en la lista de bots activos. El código responsable es el siguiente:

```
def register_bot(self):
    print("Launched a thread to listen for incoming bot subscriptions. . .")

    while True:
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock_recv:
```

```
                sock_recv.bind(("0.0.0.0", 50003))
                data, addr = sock_recv.recvfrom(1024)
                data = data.decode("utf-8")

            with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock_send:
                sock_send.sendto(b"200", (addr[0], int(data)))
                self.bot_list.append(f"{addr[0]}:{data}")

                print(f"Added new bot to active bot list: {addr[0]}:{data}")
        except:
            continue
```

Como podemos apreciar, la ejecución de este hilo tiene dos fases. En la primera, el servidor se queda escuchando en el puerto 50003 de forma bloqueante a recibir algún paquete UDP de los bots.

Una vez lo recibe, cierra ese socket, abre un segundo socket y le envía un paquete UDP de confirmación al bot que se acaba de registrar. Seguidamente, añade la IP y el puerto en el que estará el bot esperando comandos a la lista de bots en activo.

Si en cualquier momento salta alguna excepción debido a algún error de decodificación de los paquetes UDP, errores en la conexión de los sockets, etc. simplemente se hará un *continue* para seguir dentro del bucle infinito y evitar que este hilo deje de funcionar en ningún momento.

El siguiente paso en el flujo de ejecución del servidor sería el lanzamiento del hilo encargado de hacer el *stream* de Twitter, que se está ejecutando en paralelo tal y como se ha visto en el constructor.

Cada vez que se reciba un tweet que coincida con el criterio de búsqueda, se llamará al método *on_status( )*, como ya hemos anticipado anteriormente. A este método se le pasará como parámetro un objeto de tipo *status* que representa el tweet recogido por Tweepy. El código de este método es el siguiente:

```
def on_status(self, status):
    if not hasattr(status, "retweeted_status"):
        self.send_command(f"{status.user.screen_name}|{status.id}")
```

Como podemos apreciar, es bastante sencillo. Simplemente comprueba que el tweet que ha recuperado no es un retweet, y en caso de que no lo sea se llama al método encargado de enviar el comando a la botnet, creando el comando con el siguiente formato:

handle_del_usuario_que_publica_el_tweet | id_del_tweet

Los comandos deben tener este formato, debido a que para recomponer la URL del tweet hacen falta esos dos datos.

En cuanto al código del método responsable de enviar los comandos una vez se reciben, es el siguiente:

```
def send_command(self, command):
        print(f"Sending command: {command}")

        with (socket.socket(socket.AF_INET, socket.SOCK_DGRAM)) as sock:
            for bot in self.bot_list:
                bot = bot.split(":")
                sock.sendto(command.encode("utf-8"), (bot[0], int(bot[1])))
```

Simplemente se abre un nuevo socket UDP, y para cada bot registrado en la lista de bots en activo se envía un mensaje UDP a la IP:Puerto de cada bot con el comando (tweet) que deberán botear.

4.2.2. Código de los bots

Pasamos a explorar el código que implementa la funcionalidad de los bots individuales, que hacen las veces de cliente en el esquema cliente-servidor del sistema. Como ya hemos anticipado anteriormente, se ha hecho uso de la librería Selenium para automatizar las acciones en la plataforma Twitter.

Este fichero de código necesita de un fichero de configuración que le sea pasado por parámetro con las credenciales de la cuenta que usará en Twitter. Lo siguiente es un ejemplo de un fichero de configuración con el formato correcto:

```
example@email.com
account_handle
password
```

Debido a que especificar este fichero como parámetro es necesario para el funcionamiento del bot, lo primero que el bot hace es comprobar que efectivamente se le ha proporcionado como parámetro:

```
if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("Error, you need to pass a proper configuration file as an argument.")
        print("Usage: python test_client.py bot1.cfg")
    else:
        BotClient(sys.argv)
```

En caso de que el número de argumentos no coincida con el número que debe ser (2), se le enseña al usuario un mensaje con el uso correcto y se para la ejecución. Si el usuario ha proporcionado el argumento correctamente, se instancia la clase y se comienza su flujo de ejecución.

Una vez se instancia la clase, lo primero que hará será instanciar todos los parámetros de configuración del bot.

```
def __init__(self, argv):
    self.config_file = argv[1]
    self.bot_ip = "0.0.0.0"
    self.bot_port = random.randint(5000, 9999)

    self.c2_ip = "127.0.0.1"
    self.c2_port = 50003

    self.sleep_time = 1
    self.reduced_sleep_time = 0.2
    self.twitter_reply = "Muy buen tweet, estoy de acuerdo!"

    self.init_chrome_driver()
    self.twitter_login()

    self.connect_to_c2()
    self.wait_for_commands()
```

Inicialmente, se recupera el archivo de configuración de la lista de argumentos. Una vez ha hecho esto, se configura la IP y puerto tanto del bot como del servidor de *Command & Control*.

En cuanto al puerto que utilizará el bot, esté será un valor aleatorio entre 5000 y 9999, para permitir que varios bots puedan ejecutarse en la misma máquina y con el mismo fichero de código sin necesidad de estar modificando el código fuente.

Una vez se han configurado las IPs y los puertos, se configuran los tiempos de *sleep* que tendrá el bot de cara a intentar evitar la detección por parte de Twitter, así como el mensaje que los bots comentarán en los tweets que el servidor C2 haya escogido botear.

Con esto ya se han configurado todos los parámetros necesarios para el correcto funcionamiento del bot, así que lo siguiente que hará será inicializar el *driver* de Google Chrome [21].

```
def init_chrome_driver(self):
        options = Options()
        options.add_experimental_option("excludeSwitches", ["enable-logging"])
        #options.add_argument("--headless")
        self.browser = webdriver.Chrome("./chromedriver.exe", options=options)
```

El *webdriver* de Chrome es un ejecutable necesario para que el bot pueda ejecutarse, y debe coincidir con la versión de Chrome que se tiene instalada para su correcto funcionamiento. Volviendo al código, se deben configurar las opciones del *driver* quitando la opción de *logging*, para evitar que se llena la salida estándar de la terminal con todos los mensajes del log.

Además, para la versión final se tendría que descomentar la línea de código que añade la opción encargada de hacer que el *webdriver* se ejecute en modo *headless*, esto es, que se ejecute sin la interfaz gráfica de Chrome, dejando que Selenium lo gestione todo sin necesitar tener la interfaz gráfica del navegador abierta.

Una vez se ha inicializado el *webdriver*, Selenium ya puede empezar a interactuar con Chrome. Lo primero que deberá hacer el bot será iniciar sesión en Twitter:

```
def twitter_login(self):
        email, username, password = self.read_config_file()

        print("Bot attempting to log in to Twitter. . .")

        self.browser.get("https://www.twitter.com/login")
        time.sleep(self.sleep_time)

        username_input = WebDriverWait(self.browser, 10).until(expected_conditions.vi
sibility_of_element_located((By.NAME, "session[username_or_email]")))
        password_input = WebDriverWait(self.browser, 10).until(expected_conditions.vi
sibility_of_element_located((By.NAME, "session[password]")))
```

Para iniciar sesión en Twitter lo primero que hará será recuperar las credenciales del fichero que el programa ha recibido por parámetro. El método responsable de hacer esto simplemente se encarga de leer un fichero de tres líneas, por lo que su código será obviado en esta explicación.

Una vez tiene las credenciales cargadas en memoria, accede con el navegador a la página de login de Twitter.

Llegados a este punto, se deberá localizar el input del formulario dentro del código HTML de la página, para que Selenium pueda introducir las credenciales y enviarlas a Twitter. Para ello, se va a realizar una búsqueda para encontrar el *div* que tenga de nombre “session[username_or_email” y “session[password]”.

Además, se ha hecho uso de varias funcionalidades de la API de Selenium para evitar que haya fallos por inconsistencias a la hora de cargar la página. Esto se debe a que es posible, y de hecho bastante común, que se intente acceder a un elemento antes de que esté se haya cargado en la página, lo que normalmente provocaría un fallo crítico en la aplicación, forzándolo a abortar su ejecución.

Para evitar esto, se puede hacer uso de la clase *WebDriverWait*, que permite que el navegador se quede esperando hasta que se cumpla cierta condición que el desarrollador puede definir. En el caso de este código, se esperará a que el elemento con el se quiere interactuar sea visible, es decir, que se haya cargado en la página, permitiendo definir timeouts por si se ha cometido algún error.

```
username_input = WebDriverWait(self.browser, 10).until(expected_conditions.visibility
_of_element_located((By.NAME, "session[username_or_email]")))
```

Analizando la línea anterior, podemos ver que estamos intentando recuperar el campo de input para el nombre de usuario en el login. Se define que el *driver* esperará hasta 10 segundos a que el

elemento cuyo nombre es “session[username_or_email” sea visible, es decir, que haya cargado en la página.

Cada vez que se tiene que interactuar con un elemento del código HTML de Twitter se hará uso de este tipo de esperas con condición.

Volviendo a la explicación del código, una vez se hayan recuperado los campos de input, el bot deberá introducir las credenciales en estos campos y enviarlas pulsando la tecla enter:

```
        username_input.send_keys(email)
        password_input.send_keys(password)
        password_input.send_keys(Keys.ENTER)
        time.sleep(self.sleep_time)
```

Con esto, en el caso general ya se habrá hecho login y se habrá redirigido al usuario a la página principal de Twitter, esto es, a la URL *twitter.com/home*. Sin embargo, cuando Twitter empieza a detectar actividad inusual en las cuentas, pone una segunda pantalla de login en el que se deberá introducir el nombre de la cuenta, como se puede ver en la siguiente figura:

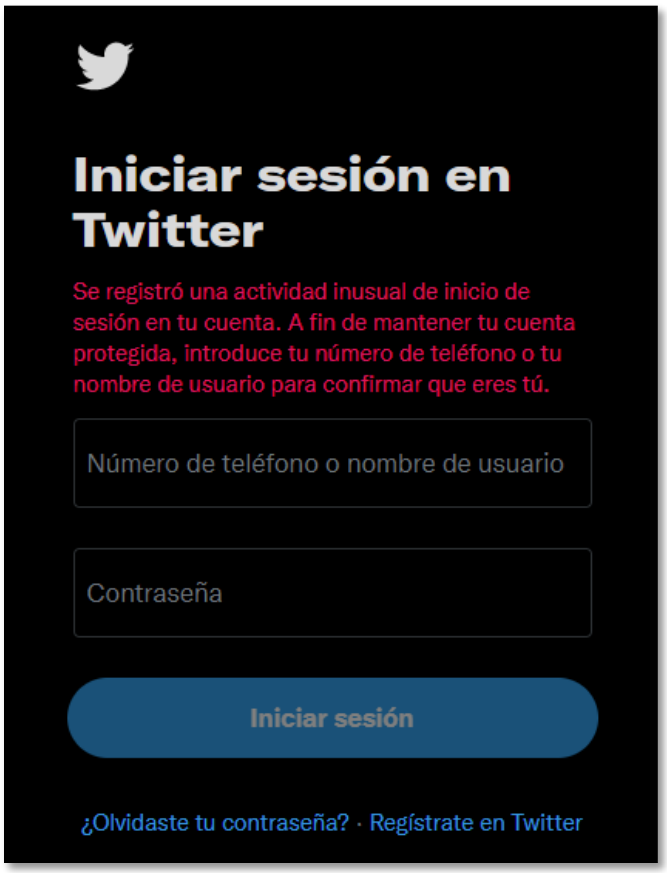

Figura 28. Mensaje de advertencia por actividad de inicio de sesión inusual.

Por ello, se debe comprobar si el navegador ha hecho la redirección a la página home porque, en caso contrario, significará que se encuentra en la segunda pantalla de login y deberá volver a realizar las acciones descritas anteriormente.

```
        if self.browser.current_url != "https://twitter.com/home":
            username_input = WebDriverWait(self.browser, 10).until(expected_condition
s.visibility_of_element_located((By.NAME, "session[username_or_email]")))
            password_input = WebDriverWait(self.browser, 10).until(expected_condition
s.visibility_of_element_located((By.NAME, "session[password]")))

            username_input.send_keys(username)
            password_input.send_keys(password)
            password_input.send_keys(Keys.ENTER)
            time.sleep(self.sleep_time)


        print(f"Bot succesfully logged in to Twitter with account: {email}")
```

Una vez haya hecho las autenticaciones necesarias, el bot quedará logeado en Twitter y podrá continuar con su ejecución.

Lo siguiente que hará será intentar conectarse con el servidor de C2, a partir de las opciones de configuración que se habían especificado en el constructor. El código es el siguiente:

```
def connect_to_c2(self):
    print("Attempting to connect to C2 server. . .")
    while True:
        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock_send:
            sock_send.sendto(str(self.bot_port).encode("utf-
8"), (self.c2_ip, self.c2_port))

        with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sock_recv:
                sock_recv.settimeout(5)
                try:
                    sock_recv.bind((self.bot_ip, self.bot_port))
                    msg, _ = sock_recv.recvfrom(1024)

                    if msg.decode("utf-8") == "200":
                        print("Succesfully connected to C2 server. . .")
                        break

                except:
                    continue
```

Para ello, se quedará en un bucle infinito hasta que consiga conectarse. Dentro de este bucle estará enviando el puerto generado aleatoriamente como mensaje al servidor C2, hasta que reciba una respuesta.

Debido a que UDP no es un protocolo orientado a conexión, no se puede saber si ha recibido el primer mensaje correctamente debido a que no hay un ACK, por lo que lo único que se puede hacer es suponer que se ha entregado correctamente y continuar con la ejecución. Ya que el

servidor debe confirmar que efectivamente ha registrado al bot en la lista de bots en activo, el bot abrirá un socket UDP esperando un mensaje de confirmación, pero cómo es posible que el servidor no haya recibido el mensaje, se deberá establecer un timeout en el socket de 5 segundos. Si pasados 5 segundos no recibe confirmación, volverá a enviar el mensaje al servidor C2.

Cuando recibe el mensaje de confirmación, saldrá del bucle infinito y podrá continuar con la ejecución, entrando en el método encargado de quedarse esperando a que lleguen comandos desde el servidor C2. El código es el siguiente:

```
def wait_for_commands(self):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((self.bot_ip, self.bot_port))

    print("Listening for incoming commands from C2 server. . .")

    while True:
        data, addr = sock.recvfrom(1024)
        data = data.decode("utf-8")
        print("Command received: ")
        print(f"rcv: {data} from {addr}")
        self.execute_command(data)
```

El código es bastante sencillo, el bot abrirá un socket UDP y se quedará escuchando en un bucle infinito las conexiones entrantes. Cuando el servidor C2 envíe un comando, se llamará al método encargado de ejecutarlo.

El código responsable de ejecutar los comandos es el siguiente:

```
def execute_command(self, command):
    command = command.split("|")
    tweet_url = f"https://twitter.com/{command[0]}/status/{command[1]}"
    self.browser.get(tweet_url)
    print(f"Bot opening status: \"{tweet_url}\"")
    time.sleep(self.sleep_time)
```

Lo primero que hará este método será recomponer la URL del tweet a través del comando, y acceder a ella a través del *webdriver* de Chrome, teniendo en cuenta que el bot ya ha iniciado sesión previamente en la plataforma.

```
    like_button = WebDriverWait(self.browser, 10).until(expected_conditions.element_t
o_be_clickable((By.XPATH, "//div[@aria-label='Like']")))
    like_button.click()
    print("Bot is liking the status. . .")

    time.sleep(self.reduced_sleep_time)

    retweet_button = WebDriverWait(self.browser, 10).until(expected_conditions.elemen
t_to_be_clickable((By.XPATH, "//div[@aria-label='Retweet']")))
```

```
    retweet_button.click()
    print("Bot is retweeting the status. . .")
    confirm_retweet_button = WebDriverWait(self.browser, 10).until(expected_condition
s.element_to_be_clickable((By.XPATH, "//div[@data-testid='retweetConfirm']")))
    confirm_retweet_button.click()

    time.sleep(self.reduced_sleep_time)

    comment_button = WebDriverWait(self.browser, 10).until(expected_conditions.elemen
t_to_be_clickable((By.XPATH, "//div[@aria-label='Reply']")))
    comment_button.click()
    print("Bot is replying to the status. . .")
    time.sleep(self.reduced_sleep_time)
    tweet_text = self.browser.find_elements_by_css_selector("br[data-text='true']")
    tweet_text[0].send_keys(self.twitter_reply)
    tweet_button = WebDriverWait(self.browser, 10).until(expected_conditions.element_
to_be_clickable((By.XPATH, "//div[@data-testid='tweetButton']")))
    tweet_button.click()

    print("Democracy successfully destroyed!")



    time.sleep(self.sleep_time)
```

Una vez ha hecho esto y se encuentra en la página del tweet, irá recuperando los botones de “Like”, “Retweet” y comentario, y los irá pulsando en ese orden. Para el comentario también se encargará de encontrar el cuadro de texto donde hacer el input del comentario, y escribir el comentario definido en el constructor de la clase.

Una vez ha hecho todo esto, el bot habrá terminado con su ejecución del comando y podrá volver a quedarse escuchando a nuevos comandos del servidor C2.

5. Conclusiones

La API de desarrolladores de Twitter proporciona una herramienta muy poderosa y sencilla de utilizar a aquellos usuarios que quieran automatizar tareas sencillas en la red social, y posibilitando el desarrollo de librerías como Tweepy para permitir desarrollar código en lenguajes de programación de alto nivel que sea capaz de interactuar con la plataforma fácilmente.

Aun así, se debe remarcar el gran número de complicaciones y limitaciones que trae consigo esta API de desarrolladores, prácticamente forzando al usuario a pagar por las versiones premium a la mínima que quiera hacer algún tipo de búsqueda compleja.

Como conclusión, se cree que la API de desarrolladores permite la creación de aplicaciones sencillas para automatizar ciertas tareas como pueden ser la respuesta a Tweets, la búsqueda rápida de Tweets recientes, etc., pero se queda muy atrás si se desean lanzar consultas que la documentación de la API considera "premium", habiendo soluciones mejores para esto como puede ser la herramienta Twint.

En cuanto a la parte del proyecto relacionado con la creación de una red de bots en Twitter, mediante el desarrollo de la prueba de concepto se ha podido demostrar lo sencillo que es desarrollar una botnet simple pero fácilmente escalable, a través de herramientas que, aunque técnicamente están prohibidas en la plataforma, pueden ser utilizadas de formar que reduzcan la capacidad que tiene Twitter de detectar este tipo de automatización de tareas.

Con esto se llega a la conclusión de que desarrollar y mantener una red de bots en Twitter es un factor especialmente peligroso debido a la penetración que tiene el uso de esta red social en la sociedad actual, posibilitando que actores externos a la plataforma mediante el uso de estas herramientas de automatización sean capaces de generar corrientes de opinión respecto a temas de actualidad como pueden ser la política, religión, etc., desvirtuando totalmente la idea original de estas redes sociales en cuanto a permitir a sus usuarios dar y recibir opiniones.

En cuanto a la compleción de los objetivos propuestos, la idea principal de la que se partía inicialmente era la creación de una herramienta que permitiese a un usuario cualquiera lanzar consultar personalizadas sobre la red social Twitter, permitiéndole recuperar y guardar los resultados de dicha consulta. En ese sentido, se cree que se ha cumplido este objetivo inicial, creándose además una serie de interfaces y herramientas para que los usuarios puedan interactuar de forma cómoda con el sistema.

De cara al segundo objetivo de este proyecto, se proponía hacer un estudio de las botnets en la red social Twitter, con el objetivo de crear una prueba de concepto de botnet capaz de replicar las acciones de una red real. De nuevo, se cree que se ha completado este objetivo con creces, habiendo implementado una infraestructura de bots capaces de recibir comandos desde un servidor C2, y realizando todas las acciones habituales de los bots de la red social.

Además, gracias a la modularidad de ambos sistemas, si en un futuro se viese conveniente añadir nuevas funcionalidades, se podría hacer fácilmente.

6. Trabajo futuro

El proyecto, aunque haya cumplido con los dos objetivos iniciales de los que se partía en cuanto a la creación de una aplicación capaz de interactuar con la plataforma Twitter, así como el desarrollo de una prueba de concepto de una botnet, podría ser ampliamente extendido a través de la implementación de funcionalidades nuevas.

En primer lugar, en cuanto a la aplicación de escritorio, sería conveniente revisar la implementación actual de las búsquedas avanzadas en la plataforma, incrementando la funcionalidad actual y permitiendo hacer búsquedas por geolocalización y otros parámetros.

Además, para suplir las deficiencias del modo estándar de la API premium, se debería cambiar la implementación de este tipo de búsquedas, por la de la librería Twint o, al menos, implementar un modelo mixto que permita hacer búsquedas con una o con otra librería.

Por último, las interfaces gráficas asociadas a las aplicaciones son muy sencillas y, por lo tanto, bastante planas. Estas interfaces podrían ser enriquecidas para ser más llamativas para sus usuarios.

En cuanto a la parte del proyecto relacionada con la botnet, todo el trabajo futuro en el que se ha pensado sería extender la funcionalidad ya existente, pudiendo implementar las siguientes funcionalidades:

- Crear un sistema para automatizar la creación de cuentas de Google.
- Una vez creado el sistema anterior, crear un sistema para la automatización de cuentas de Twitter, utilizando los correos de Google generados anteriormente, y guardando las credenciales en ficheros de configuración utilizables por los bots.

Con estas dos nuevas funcionalidades se tendría una botnet capaz de generar nuevas cuentas de usuario utilizables por sus bots de forma automática para que, en el caso de que alguna de las cuentas fuese baneada por actividad inusual, se pudiese usar una de las cuentas creadas.

El objetivo de la botnet no es infectar máquinas y replicarse automáticamente, debido a que no está pensada para funcionar como un malware. En caso de que el usuario quisiese darle ese toque de malware de auto propagación, se podrían crear sistemas de replicación automática a través de la explotación de malas configuraciones en servidores de todo tipo, o a través de la explotación de vulnerabilidades como hacen las botnets convencionales.

En ese caso sería necesario adoptar medidas para evitar la detección, como podría ser la ofuscación del código, y, antes de nada, compilar el código para evitar que sea un script interpretado legible por cualquier usuario.

Por último, se podría mejorar la infraestructura de red para soportar funcionalidades como pueden ser detectar cuando ciertos bots hayan muerto para eliminarlos de las listas de bots en activo, tener

algún sistema para gestionar colas de comandos por si el usuario define términos de búsqueda que recuperen muchos resultados, etc.

7. Coste del proyecto

El presupuesto para el presente proyecto se divide en el coste de la mano de obra y el coste de materiales e infraestructura.

En cuanto al coste de la mano de obra:

Tabla 1. Desglose de costes de mano de obra del proyecto.

CONCEPTO	HORAS	COSTE/HORA	COSTE TOTAL
INGENIERÍA			
Jefe de proyecto	25	32	800
Analista Programador	40	20	800
Programador	150	14	2.100
SECRETARÍA			
Secretaría	120	12	1.440
TOTAL			**5.140,00**

En el concepto de INGENIERÍA se incluye:

- Análisis del sistema en función de los requisitos:
    - Diseño y desarrollo de las partes que componen el sistema.
    - Diseño y desarrollo de las interfaces de usuario de las aplicaciones.
    - Estudio e implementación de los mecanismos necesarios para interactuar con la red social Twitter de forma automatizada.
- Depuración del código y pruebas sobre el sistema.
- Creación de la documentación del código.

En el concepto de SECRETARÍA se incluye:

- Redacción de la memoria.
- Creación de documentos e informes adicionales sobre el desarrollo del proyecto.

En cuanto al coste de los materiales, podemos hacer una diferenciación entre el coste de la infraestructura hardware, y el coste de las licencias software necesarias para el desarrollo del proyecto.

Tabla 2. Desglose del coste hardware.

CONCEPTO	UNIDADES	COSTE/UNIDAD	COSTE TOTAL
Ordenador de sobremesa, Ryzen 7 5800X 3.8GHz, 512 GB SSD, tarjeta gráfica Nvidia RTX 3070 8 GB GDDR6, 32GB memoria DDR4	1	1.830,00	1.830,00
Monitor de 24" AOC	2	120	240
TOTAL			**2.070,00**

Tabla 3. Desglose del coste de licencias software.

CONCEPTO	NÚMERO DE LICENCIAS	COSTE/UNIDAD	COSTE TOTAL
Windows 10	1	116,99	116,99
Microsoft Office	1	69,90	69,90
Toad Data Modeler	1	618,31	618,31
Visual Studio Code	1	0	0
TOTAL			**805,20**

Tabla 4. Coste total de materiales.

CONCEPTO	COSTE TOTAL
COSTE TOTAL HARDWARE	2.070,00
COSTE TOTAL DE LICENCIAS SOFTWARE	805,20
COSTE TOTAL DE MATERIALES	**2.875,20**

Por último, se calculará el coste de los gastos generales. En este apartado se incluyen gastos como:

- Material de oficina (folios, bolígrafos, etc.).

- Dietas y desplazamientos.
- Gastos en infraestructura (luz, conexión a Internet, etc.).

Para calcular el coste de los gastos generales se aplicará un recargo del 20% sobre la suma del total de los costes materiales y el total de los costes de mano de obra.

Tabla 5. Gastos generales del proyecto.

CONCEPTO	**COSTE TOTAL**
GASTOS GENERALES	**1.603,04**

Con todo esto, nos queda el siguiente desglose del coste global del proyecto:

Tabla 6. Desglose del coste global del proyecto.

CONCEPTO	**COSTE TOTAL**
COSTE TOTAL DE MANO DE OBRA	5140,00
COSTE TOTAL DE MATERIALES	2.875,20
GASTOS GENERALES	1.603,04
COSTE TOTAL DE MATERIALES	**9.618,24**

El coste global del proyecto asciende a la cantidad de NUEVE MIL SEISCIENTOS DIECIOCHO EUROS CON VEINTICUATRO CÉNTIMOS.

8. Bibliografía

[1] Smart Insights - Global social media statistics research summary 2021: https://www.smartinsights.com/social-media-marketing/social-media-strategy/new-global-social-media-research/

[2] Statista – Social media – Statistics & Facts: https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/

[3] Backlinko – Twitter Users: https://backlinko.com/twitter-users

[4] Simplilearn – Understanding the Impacts of Social Media: https://www.simplilearn.com/real-impact-social-media-article

[5] Guido Caldarelli, Rocco De Nicola, Fabio Del Vigna, Marinella Petrocchi & Fabio Saracco – The role of bot squads in the political propaganda on Twitter: https://www.nature.com/articles/s42005-020-0340-4

[6] Tweepy – Documentation reference: https://docs.tweepy.org/en/stable/index.html

[7] Twitter – Developer Platform: https://developer.twitter.com/en

[8] Wikipedia – Twitter: https://en.wikipedia.org/wiki/Twitter

[9] Twitter Developer – Authentication: https://developer.twitter.com/en/docs/authentication/overview

[10] Scrapy – Scrapy framework documentation: https://scrapy.org/

[11] Github – Twint Twitter Intelligence Tool – https://github.com/twintproject/twint

[12] Selenium – Non-official documentation: https://selenium-python.readthedocs.io

[13] Twitter help – Rules & policies – Twitter automation: https://help.twitter.com/en/rules-and-policies/twitter-automation

[14] Twitter blog – Automation and the use of multiple accounts: https://blog.twitter.com/developer/en_us/topics/tips/2018/automation-and-the-use-of-multiple-accounts

[15] Onur Varol, Emilio Ferrara, Clayton A. Davis, Filippo Menczer, Alessandro Flammini – Online Human-Bot Interactions: Detection, Estimation, and Characterization: https://aaai.org/ocs/index.php/ICWSM/ICWSM17/paper/view/15587/14817

[16] Python docs – tkinter: https://docs.python.org/3/library/tkinter.html

[17] Kivy – Cross-platform Python Framework for NUI Development: https://kivy.org/#home

[18] Qt framework – Cross-platform software development for embedded & desktop: https://www.qt.io/

[19] Qt Documentation – Qt Designer: https://doc.qt.io/qt-5/qtdesigner-manual.html

[20] Twitter Documentation – Stream API: https://developer.twitter.com/en/docs/tutorials/stream-tweets-in-real-time

[21] Chrome Driver – Downloads: https://chromedriver.chromium.org/downloads

[22] Herramienta utilizada para crear los diagramas: https://app.diagrams.net/

9. Anexo A – Elementos adicionales entregables

Este anexo recoge una descripción de los elementos que no son entregables directamente con el proyecto, pero que se apreciaría que se tuviesen en cuenta a la hora de evaluar el mismo.

Para entregar estos elementos, se ha creado un repositorio de GitHub público que se puede encontrar en el siguiente enlace: https://github.com/EduardoGravan/TFM

El repositorio de GitHub ha sido utilizado como forma de almacenar las versiones en una plataforma online a modo de copia de seguridad y control de versiones. Se ha estado utilizando durante todo el proyecto, por lo que se puede utilizar como histórico del mismo a través de la revisión de los *commits*.

Los contenidos que se podrán encontrar en este repositorio son:

- Código fuente de la aplicación de escritorio interactuar con Twitter. La estructura de este directorio sigue la estructura básica de un proyecto Maven creado con Apache Netbeans.

- Código fuente de la prueba de concepto de botnet creada, tanto el cliente como el servidor C2.

- Dentro del directorio resources, se encuentra la carpeta de la base de datos SQLite. Se entrega una versión con la base de datos inicializada y cargada con datos, así como los scripts SQL necesarios para crear las tablas. Para automatizar la ejecución de este script SQL, se ha creado un script en formato .bat.

- Directorio de documentación. En este directorio se podrá encontrar la memoria entregada y la presentación PowerPoint utilizada en la defensa.

En el caso de querer probar las aplicaciones, es importante seguir las recomendaciones del fichero *README.md* localizado en la raíz del repositorio, esto es, crear los ficheros con las claves para que la aplicación sea capaz de autenticarse con la API de desarrolladores de Twitter. Obviamente, se tendrán que instalar las dependencias del proyecto listadas en el archivo *requirements.txt*.