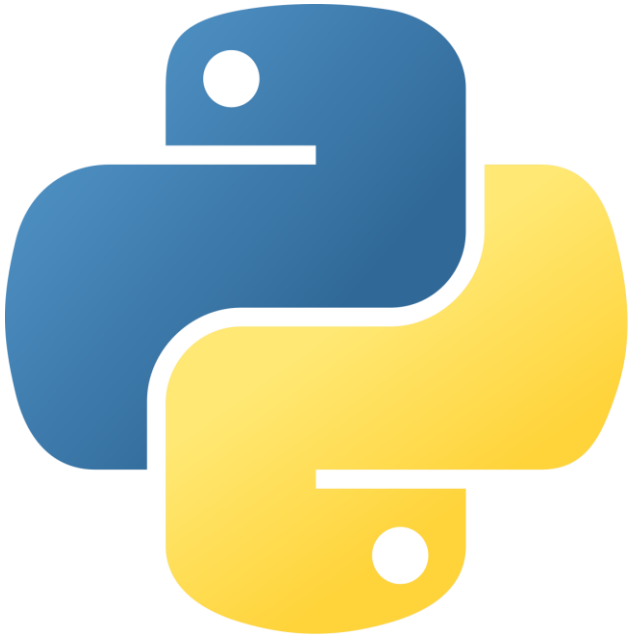

IMPLEMENTACIÓN DE UN CHAT UTILIZANDO UN MODELO CLIENTE-SERVIDOR

PYTHON PARA INGENIERÍA DE TELECOMUNICACIONES

Antonio González López

Contenido

Objetivos del proyecto

El objetivo principal de este proyecto es implementar una aplicación de comunicación en tiempo real usando un modelo cliente-servidor que permita a los usuarios intercambiar mensajes de forma segura. A continuación, se describirán estos objetivos de forma más detallada.

Se quiere diseñar una aplicación distribuida usando una arquitectura cliente-servidor. Los clientes serán los usuarios, que escribirán y leerán los mensajes, mientras que el servidor se encargará de recibir los mensajes de los clientes y reenviarlos al resto de usuarios. Para ello, se establecerá una conexión TCP/IP entre el servidor y cada uno de los clientes, ejecutando cada una de estas conexiones en un hilo. Estos hilos trabajarán en paralelo y permiten que el servidor pueda atender a varios clientes a la vez. Si solo hubiese un hilo, solo se podría crear una conexión punto a punto entre el servidor y el cliente, por lo que queda más que argumentado el uso de un servidor multihilo, ya que permite cumplir el objetivo principal de este proyecto, comunicar varios clientes.

Además, se quiere añadir encriptado simétrico a los mensajes que envían los clientes, que compartirán la misma clave. De esta forma, tan solo los clientes podrán descifrar los mensajes, ni siquiera el servidor podrá leer los mensajes originales (sin encriptar) de los clientes. En este caso, se usará el cifrado César por ser especialmente fácil de implementar.

A modo de resumen, se tienen las siguientes metas:

1. Implementar un chat funcional donde los usuarios puedan comunicarse con mensajes.
2. Utilizar una arquitectura cliente-servidor escalable que permita la comunicación entre dos o más usuarios.
3. Asegurar la privacidad y la integridad de los mensajes mediante el uso de encriptación simétrica.

Código

Antes de explicar resultados que se han obtenido, es necesario explicar el código que se ha implementado. Para todo este proyecto se han usado la librería *socket*, que permite la creación de conexiones TCP/IP entre distintos equipos y la librería *threading*, que permite la ejecución de hilos.

Aplicación servidor

Apertura del puerto

```
import socket
import threading as threading


host=socket.gethostbyname(socket.gethostname())  #IP PRIVADA DEL SERVIDOR
port=8000             #PUERTO DEL SERVIDOR DONDE SE ATIENDEN PETICIONES

socketserver=socket.socket(socket.AF_INET,socket.SOCK_STREAM)       #SE
CREA EL SOCKET
socketserver.bind((host,port)  #SE VINCULA EL SOCKET A UNA IP Y UN PUERTO
socketserver.listen()                  #EL SERVIDOR EMPIEZA A ESCUCHAR
DESDE EL PUERTO PARA ATENDER LAS CONEXIONES DEL CLIENTE
```

```
print(f"El servidor se encuentra en:
{socket.gethostbyname(socket.gethostname())}")      #SE DA IP DEL
SERVIDOR

print("Esperando conexión con clientes...")

clientes=[]             #LISTA DE CLIENTES
```

En primer lugar, se guarda en la variable *host* la dirección IP privada del servidor. Al ser una dirección privada, solo los clientes que se encuentren en la misma red podrán establecer conexión con el servidor.

A continuación, se guarda en la variable *port* el número de puerto que el servidor abrirá para poder atender las peticiones de los clientes. Se ha seleccionado el puerto 8000 porque en el ordenador que se usa como servidor no tiene ningún uso (comprobado con el comando netstat -an), y porque tiene uso como puerto TCP, ya que hay otros que son UDP.

Posteriormente, se crea un socket y se vincula a la dirección ip del servidor y su puerto, permitiendo la posterior creación de conexiones TCP/IP. El socket comienza a escuchar las peticiones. Se imprime por pantalla la dirección IP del servidor, que se comparte con los usuarios de forma física; es decir, se debe enseñar la dirección IP a los usuarios para que la puedan escribir al ejecutar la aplicación cliente, como posteriormente se explicará en la explicación del código Cliente.py. Se ha elegido hacerlo de este modo porque la dirección ip del servidor cambia debido a dos razones: depende de la dirección de red y, en el caso de que estuviera siempre en la misma red, tendría una dirección variante, debido a la acción del protocolo DHCP.

Se crea la lista *clientes*, que almacenará los distintos clientes que se conecten al servidor.

Recibir y reenviar mensajes de clientes

El principal objetivo del servidor es recibir los mensajes que envía cada cliente y enviarlos al resto de clientes. Se han diseñado dos funciones que nos permitirán hacerlo: *broadcast( )* y *recibiryenviar( )*.

En primer lugar, se analizará la función *broadcast( )*, que como indica su nombre, envía el mensaje a todos los dispositivos conectados a la red (exceptuando al cliente que lo ha enviado previamente).

```
def broadcast(msj,remitente):           #FUNCIÓN QUE ENVÍA MENSAJE A
TODOS LOS CLIENTES MENOS A REMITENTE
    for client in clientes:
        if client != remitente:         #SI EL CLIENTE NO ES EL REMITENTE
            try:
                client.sendall(msj.encode("utf-8"))         #SE ENVÍA EL
MENSAJE
            except:
                clientes.remove(client)                     #SI HAY ALGÚN
ERROR, SE ELIMINA AL CLIENTE DE LA LISTA DE CLIENTES
                print(f"Desconectando a {address} por un error en la
conexión")
```

Los parámetros de entrada de esta función son *msj* (mensaje que se quiere enviar) y *remitente*, que es el cliente que ha enviado el mensaje. Se compara el remitente con cada uno de los elementos de la lista *clientes* (recordemos que en esta lista se almacenarán todos los clientes con los que se haya creado una conexión). Solo en caso de que sean distintos se enviará el mensaje, para evitar que aparezca un mensaje duplicado en la pantalla del cliente remitente. El mensaje es enviado usando utf-8, es decir, se transforman en bytes los caracteres alfanuméricos introducidos por el usuario. Si ocurre una excepción, como que no se pueda enviar el mensaje al cliente algún motivo relacionado al hilo de ejecución o a uno de los sockets del cliente y del servidor, se elimina este cliente de la lista de clientes y se cierra la conexión.

A continuación, se desarrolla la explicación del código de la función *recibiryenviar( )*, que se encarga de recibir los mensajes de los clientes y llamar a la función *broadcast( )* cuando sea necesario.

```
def recibiryenviar(client,address):     #FUNCIÓN QUE ESTABLECE CONEXIÓN
CON UN CLIENTE Y SE ENCARGA DE DISTRIBUIR EL TRÁFICO

    clientes.append(client)             #SE AÑADE EL NUEVO CLIENTE A LA
LISTA DE CLIENTES
    print(f"Conexión establecida desde {address}")

    while True:                         #BUCLE INFINITO
        try:
            msj=client.recv(1024).decode("utf-8")           #SE RECIBEN
LOS MENSAJES ENCRIPTADOS DEL CLIENTE (MÁXIMO 1024 BYTES)
            if msj:                                         #SI HAY UN
MENSAJE
                print(f"Mensaje recibido desde {address}:
{msj}")       #SE IMPRIME EN PANTALLA EL MENSAJE ENCRIPTADO
                broadcast(msj,client)                       #EL MENSAJE
RECIBIDO POR EL SERVIDOR SE ENVÍA A TODOS LOS CLIENTES
        except:
            broadcast(f"{client} se ha desconectado", client)
            print(f"Conexión cerrada desde {address}")
            clientes.remove(client)         #EN CASO DE ERROR SE AVISA DE
LA DESCONEXIÓN Y SE ELIMINA AL CLIENTE DE LA LISTA
            break                           #SE ACABA EL BUCLE
```

Los parámetros que se introducen en esta función son: el cliente desde el que se ha enviado el mensaje que llega al servidor y la dirección IP del cliente.

En primer lugar, una vez se ha aceptado la conexión entre el servidor y el cliente, se incluye el cliente en la lista *clientes*, que es fundamental para el correcto funcionamiento del resto de funciones. Se crea un bucle infinito, es decir, cuando se ejecute esta función, permanecerá en ejecución hasta que salte una excepción y haya un *break* que rompa el bucle. El servidor estará a la espera de recibir un mensaje del cliente (estará encriptado). Si no se recibe el mensaje, no

se ejecutarán las siguientes líneas de código y volverá a buscar un mensaje hasta que le llegue alguno. Se recibe el mensaje, que ocupará como máximo 1024 bytes, se convierte en texto usando el comando decode("utf-8") y se usa envía a todos los clientes usando la función *broadcast( )* que se ha desarrollado antes.

En caso de que haya alguna excepción (p. ej. error en conexión de sockets), se avisa al resto de usuarios de que el cliente se ha desconectado y se elimina a este cliente de la lista *clientes*, cerrando así su conexión con el servidor. Aparece el *break* que se ha mencionado anteriormente y se termina de ejecutar la función.

Hilos y ejecución de la función *recibiryenviar( )*

Ahora que se ha definido el uso de la función *recibiryenviar( )*, que será la función más importante del código, es necesario saber cómo se hace la llamada.

```
while True:                 #POR SIEMPRE
    client,address=socketserver.accept()            #ACEPTAR LAS
CONEXIONES QUE LLEGAN AL SOCKET
    hilo =
threading.Thread(target=recibiryenviar,args=(client,address))        #SE
CREA UN HILO POR CADA CLIENTE QUE EJECUTA LA FUNCION RECIBIRYENVIAR()
    hilo.start()            #SE EJECUTA EL HILO
```

El bucle infinito permite que el servidor esté siempre en ejecución y permita a los clientes mandar mensajes hasta que se cierre la conexión. La función *socketserver.accept( )* acepta la solicitud de conexión que llega al puerto del servidor. En la variable *client* se guarda el cliente que ha solicitado la conexión y en la variable *address* su IP.

Sería necesario ejecutar una función *recibiryenviar( )* por cada cliente. Esto se consigue creando un hilo en el servidor para cada cliente que se conecta al puerto de escuchay que se encarga de ejecutar la función *recibiryenviar( )*, con el cliente y la su dirección IP como argumentos de entrada. Con la función *start( )*, el hilo comienza a ejecutarse.

Aplicación cliente

Conexión con servidor

En primer lugar, se debe establecer la conexión TCP/IP con el servidor.

```
import socket
import threading

ipserver=input("Escribe la IP del servidor: ")
port=8000                   #PUERTO DEL SERVIDOR QUE ATENDERÁ PETICIONES

#SE CREA CONEXIÓN TCP/IP ENTRE CLIENTE Y SERVIDOR
client=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client.connect((ipserver,port))
nombre=input("Escribe tu nombre de usuario: ")      #SE PREGUNTA A
USUARIO SU NOMBRE
```

Se pregunta al usuario cuál es la dirección IP del servidor y la guarda en la variable *ipserver*. Por los motivos expuestos anteriormente, el administrador del servidor deberá proporcionar la IP. El puerto donde el servidor escucha las comunicaciones de los clientes es el 8000, y se guarda en la variable *port*.

A continuación, se crea un socket llamado *cliente* y se conecta con la dirección IP guardada en *ipserver* y con el puerto 8000. Es decir, el socket del cliente intenta iniciar la conexión con el socket del servidor.

Posteriormente, se pregunta el nombre de usuario que se va a usar.

Encriptación y desencriptación de mensajes con cifrado César

Se crean las funciones *encriptar( )* y *decrypt( )*.

```
def encriptar(mensaje, desplazamiento):           #ENCRIPTACIÓN CIFRADO
CÉSAR
    mensaje_encriptado = ""
    for caracter in mensaje:
        if str(caracter).isalpha():
            ascii_inicial = ord('a') if str(caracter).islower() else
ord('A')
            ascii_encriptado = (ord(str(caracter)) - ascii_inicial +
int(desplazamiento)) % 26 + ascii_inicial      #FÓRMULA ENCRIPTADO
            caracter_encriptado = chr(ascii_encriptado)
            mensaje_encriptado += caracter_encriptado
        else:
            mensaje_encriptado += str(caracter)
    return mensaje_encriptado
```

A modo de resumen, el cifrado César es un cifrado de desplazamiento que consiste en cambiar cada letra del mensaje por otra distinta que esté *k* posiciones por delante en el abecedario, donde *k* es la "clave". La fórmula para hallar el carácter codificado *C*, donde *X* es el carácter sin codificar, es:

$C = (X + k) mod (26)$. Se utiliza el número 26 porque es el número de letras del abecedario. La función *mod* es el equivalente a hallar el resto de una división.

```
def decrypt(mensaje_encriptado, desplazamiento):    #DESENCRIPTACIÓN
CIFRADO CÉSAR
    mensaje_desencriptado = ""
    for caracter in mensaje_encriptado:
        if caracter.isalpha():
            ascii_inicial = ord('a') if caracter.islower() else ord('A')
            ascii_desencriptado = (ord(caracter) - ascii_inicial -
desplazamiento) % 26 + ascii_inicial
            caracter_desencriptado = chr(ascii_desencriptado)
            mensaje_desencriptado += caracter_desencriptado
```

```
        else:
            mensaje_desencriptado += caracter
    return mensaje_desencriptado
```

En cuanto a la desencriptación, es exactamente el mismo proceso que la encriptación, salvo que en lugar de cambiar el carácter por uno que esté *k* posiciones por delante en el abecedario, debe estar *k* posiciones por detrás. La fórmula para la desencriptación, usando la misma nomenclatura que para la encriptación, es:

$X = (C - k) mod (26)$.

Recepción de mensajes

```
def recibir():          #FUNCIÓN PARA RECIBIR MENSAJES DESDE EL SERVIDOR
    while True:
        try:
            msj=client.recv(1024).decode('utf-8')       #SE RECIBEN
MENSAJES DE MÁXIMO 1024 BYTES Y SE DECODIFICAN USANDO UTF-8
            print(decrypt(msj,10))     #SE DESENCRIPTA CON CIFRADO CÉSAR

        except:                 #EN CASO DE QUE HAYA ALGÚN ERROR...
            print("Desconectando del servidor...")
            client.close()
            break
```

La función *recibir( )* sirve para escuchar los mensajes que llegan desde el servidor hasta el cliente. Permanece en un bucle infinito hasta que se produce un error en la transmisión entre los sockets, cuando aparece la palabra *break* y se desconecta el cliente. En caso de que no haya error, el cliente está siempre escuchando por el socket. Cuando se recibe un mensaje, de máximo 1024 bytes, se debe transformar de bytes a texto, para ello se usa la función *decode("utf-8)*. El mensaje sigue encriptado con cifrado César, por lo que para recuperar el mensaje que quería mandar el otro cliente originalmente, se usa la función *decrypt( )*, con una clave de 10, que será el mismo número que se usará para encriptarlo cuando se envíen los mensajes (se explicará más adelante).

Ejecución del hilo y envío de mensajes

```
hilorecibir = threading.Thread(target=recibir)          #UN HILO SE
EJECUTA SIEMPRE Y SE ENCARGA DE RECIBIR MENSAJES
hilorecibir.start()                                     #SE INICIA EL
HILO


while True:                         #CLIENTE ESCUCHANDO SIEMPRE A
USUARIO PARA MANDAR MENSAJES AL SERVIDOR
    msj=input('>')                  #MENSAJE QUE ESCRIBE EL USUARIO
    if msj:                         #EN CASO DE QUE EL USUARIO HAYA
ESCRITO UN MENSAJE
        if msj.upper()=="FIN":      #SI USUARIO ESCRIBE FIN --> SE
TERMINA LA CONEXIÓN
            client.close()
            break
```

```
        else:
            msjsinencriptar=f"{nombre}: {msj}"      #SE CONCATENA EL
NOMBRE DE USUARIO CON EL MENSAJE
            msjencriptado=encriptar(msjsinencriptar,10).encode("utf-
8")     #SE ENCRIPTA EL MENSAJE USANDO CIFRADO CÉSAR CON DESPLAZAMIENTO
10
        #    print(msjencriptado)    #COMPROBACIÓN DE QUE SÍ ENCRIPTA
            client.sendall(msjencriptado)           #SE ENVÍA EL MENSAJE
ENCRIPTADO AL SERVIDOR
```

Se crea un hilo que se encarga de ejecutar la función *recibir( )* y se ejecuta. Se inicia un bucle infinito en el que se espera a que el usuario escriba un mensaje. Si el usuario ha escrito algo, se comprueba que es distinto de "FIN". En caso de que sea igual, se cierra la conexión con el servidor. Si es distinto, se concatena el nombre de usuario con el mensaje que se quiere enviar, se encripta usando cifrado César con una clave de 10 y se envía al servidor.

Resultados obtenidos

A continuación, se expondrán capturas de pantalla de la ejecución de los códigos *Servidor.py* y *Cliente.py*, demostrando su correcto funcionamiento y la interacción con el usuario.

En primer lugar, se debe desactivar el firewall para redes públicas del ordenador que va a actuar como servidor. Este paso es fundamental, porque si no, los clientes no podrán establecer la conexión con el puerto 8000.

Una vez desactivado el firewall, se procede a ejecutar el archivo *Servidor.py*.

```
El servidor se encuentra en: 192.168.0.21
Esperando conexión con clientes...
```

Se indica cuál es la dirección IP del servidor para que los clientes puedan saberla.

Se ejecuta el archivo *Cliente.py* en otro ordenador.

```
Escribe la IP del servidor:
```

Aparece un mensaje de tipo *input* solicitando la IP del servidor al usuario.

```
Escribe la IP del servidor: 192.168.0.21
Escribe tu nombre de usuario: Ordenador 1
```

Cuando el usuario ha introducido la IP del servidor, le pregunta cuál va a ser su nombre de usuario en el chat. En este caso, se ha elegido "Ordenador 1".

```
Esperando conexión con clientes...
Conexión establecida desde ('192.168.0.15', 51188)
```

En la aplicación del servidor aparece el mensaje de la imagen superior. Se indica que se ha conectado un cliente desde la IP 192.168.0.15, donde 51188 es el puerto que usa el cliente para iniciar el socket.

Se repite el proceso de conectar otro cliente. En este caso, por falta de número de equipos, se usará el mismo ordenador que el para el servidor para ejecutar el cliente "Ordenador 2". Por tanto, tendrá la misma dirección IP.

```
Conexión establecida desde ('192.168.0.21', 50356)
```

Una vez establecidas las conexiones con el servidor, las aplicaciones cliente pueden empezar a enviar mensajes. Aparece una pantalla vacía en la que se pueden escribir los mensajes.

```
>¡Hola! Soy el Ordenador 2
>
```

Desde Ordenador 2 enviamos el mensaje que aparece en la imagen superior. En la aplicación del servidor aparece de la siguiente forma:

```
Mensaje recibido desde ('192.168.0.21', 50356): Ybnoxknyb 2: ¡Ryvk! Cyi ov Ybnoxknyb 2
```

Como se puede ver, el mensaje llega al servidor completamente encriptado usando cifrado César. El servidor no descifra este mensaje para no vulnerar la privacidad de los usuarios. Simplemente se limita a enviar el mismo mensaje que le llega.

En el otro cliente (Ordenador 1) aparece lo siguiente:

```
>Ordenador 2: ¡Hola! Soy el Ordenador 2
```

Se puede ver el mensaje que Ordenador 2 ha enviado, usando la forma "{usuario}: {mensaje decodificado}". El cliente lo ha conseguido decodificar porque comparte la clave con el otro cliente.

A continuación, se muestra una conversación desde el punto de vista de Ordenador 2:

```
>¡Hola! Soy el Ordenador 2
>Ordenador 1: Yo soy Ordenador 1
>Esto es un proyecto de Python
>Ordenador 1: ¿Funciona bien?
>Sí, leo tus mensajes perfectamente
```

En resumen, se han cumplido los objetivos propuestos al inicio del proyecto. Se ha conseguido desarrollar un chat en forma de aplicación distribuida con arquitectura cliente-servidor. Se ha trabajado con un servidor multihilo y para permitir la escucha simultánea del servidor hacia los distintos clientes. De esta forma, se ha cumplido el objetivo de permitir la comunicación entre dos o más usuarios. También se ha logrado la encriptación de los mensajes enviados usando cifrado César.

Problemas encontrados

Se han encontrado cinco principales problemas durante el proceso de creación de esta aplicación:

1. Uso de librerías y consulta de distintas fuentes.
2. Búsqueda de puertos disponibles para el servidor.
3. Implementación del cifrado de mensajes.
4. Envío de los mensajes como bytes.
5. Cortafuegos.

El primer obstáculo encontrado en este proyecto fue el desconocimiento del funcionamiento de las librerías *socket* y *threading*. He usado librerías parecidas en Java, pero al no haberlas usado nunca en Python ha sido necesario leer documentación y tutoriales acerca de su uso. Uno de los problemas de consultar distintas fuentes y manuales es que cada uno implementa el código de una forma, usando cada uno un "protocolo" (por llamar de alguna forma a la manera de establecer conexiones); por tanto, es imposible copiar códigos de webs distintas porque no pueden funcionar a la vez. Para solucionar este problema, la solución ha sido uniformizar todos los códigos que he encontrado, esto es, escoger las partes de los distintos códigos que convenían y rechazar o modificar las demás.

En segundo lugar, apareció un problema bastante fácil de solucionar. Había que buscar un puerto en el ordenador que no se usara con ninguna otra aplicación y que pudiera usarse con el protocolo TCP/IP. Para buscar los puertos en uso, simplemente hay que escribir netstat –an en la terminal del ordenador. El puerto 8000 no aparecía en la lista de puertos en uso, por lo que, al usar TCP, fue uno de los puertos libres que podían ser escogidos.

Posteriormente, apareció el problema de hallar qué tipo de cifrado podría ser el adecuado para usar en el proyecto. Se eligió una encriptación simétrica, concretamente el cifrado César, por tener un algoritmo sencillo de entender y, presumiblemente, fácil de implementar. Sin embargo, fue necesario consultar el funcionamiento de algunas funciones relacionadas con el procesado de textos y ASCII. Para elaborar esta parte de código que usó la ayuda de Chat GPT.

Cuando se intentaban enviar los mensajes como una variable *string* a través del socket, aparecía un error relacionado con la necesidad de enviar mensajes de tipo byte y no texto. Para solucionar esto, los mensajes, antes de ser enviados, deben ser transformados a bytes, usando la forma *{mensaje}.encode("utf-8*"), donde UTF-8 es un formato de codificación de caracteres. Ya que se envían bytes, también es necesario que al recibir estos bytes se transformen a texto otra vez, para ello se escribe *{mensajerecibido}.decode("utf-8")*.

Por último, apareció un problema que fue más complicado de identificar. Cuando se utilizaba la aplicación de cliente en el mismo ordenador que la del servidor, ambas funcionaban correctamente; sin embargo, al utilizar el cliente en otro ordenador distinto, el servidor rechazaba la conexión. Se probaron varias formas de introducir la IP, se cambió el número de puertos y se comprobó que no fuera un problema del funcionamiento de los hilos del servidor. Después de probar, el problema era mucho menor que todo lo que se pensó, ya que, para solucionarlo, solo había que desactivar el firewall, que estaba rechazando todas las conexiones externas al puerto 8000.

Futuras mejoras

Para concluir, es necesario explicar algunas de las ideas que podrían mejorar mucho este proyecto, pero que, por falta de tiempo y recursos, han sido imposibles de implementar. Encontramos dos grandes vías de mejora: creativas y funcionales.

- Creativas. Como se puede observar en las imágenes expuestas previamente, esta es una aplicación que se ejecuta desde la terminal del ordenador, con su característico fondo negro y letras blancas. Esto hace que sea una aplicación poco atractiva visualmente. Para mejorar este aspecto, se consideró la posibilidad de usar la librería *tkinter*, que, a pesar de no dar unos resultados excesivamente vistosos, permite crear una GUI (*Graphical User Interface*) más intuitiva y fácil de usar. Esta idea no se pudo llevar a cabo porque estudiar el uso de una librería más solo habría sido posible sacrificando otros aspectos funcionales (bajo mi punto de vista, más importantes) más relacionados con las telecomunicaciones.
- Funcionales. En este aspecto han surgido muchas ideas que podrían elevar este proyecto a un nivel superior.
    - En primer lugar, conseguir una dirección IP pública permanente para el servidor permitiría que se pudiesen conectar clientes de distintas redes. Tal y como está el proyecto, solo se pueden conectar los dispositivos que se encuentran en la misma red que el servidor.
    - Otra posible mejora sería añadir la posibilidad de crear distintas salas de chat y la posibilidad de enviar mensajes privados entre clientes. En estos momentos, solo existe una única sala por servidor. Quizá indagando en el estudio de la librería *threading* se podría conseguir algo parecido, aunque es una idea demasiado complicada para haberla desarrollado en este proyecto.
    - Otra gran posible mejora sería añadir el envío de archivos e imágenes. El envío de archivos .txt no resultaría excesivamente complicado de implementar; sin embargo, el de imágenes sí que supondría un reto mayor.
    - Para finalizar, sería necesaria una encriptación de datos mucho más segura. El cifrado César es muy simple y relativamente fácil de decodificar y, aunque para este proyecto pueda servir como un ejemplo de encriptación simétrica, en el ámbito profesional apenas se usa. Se podría implementar una encriptación más compleja y segura, como por ejemplo, el *Advanced Encryption Standard (AES)*.