Universidad Nacional del Nordeste

Facultad de Ciencias Exactas y Naturales y Agrimensura

Cátedra: Sistemas Operativos Año:2024

EJERCICIO PRÁCTICO N.º 4.2

Alumno: Octavio Alfredo Sánchez - DNI:44622119

Ejercicio 1: Se requiere realizar la programación e implementación de un chat básico mediante el uso de sockets.

En este caso decidí usar el lenguaje de programación Python para resolver esta primera actividad
Explicación del funcionamiento del chat de red a nivel general

1. Aplicación de chat en red utilizando sockets

El código implementa una aplicación de chat en red utilizando sockets. Se establece una comunicación en tiempo real entre varios usuarios conectados a un servidor central mediante el uso de sockets TCP.

2. Comunicación entre al menos dos usuarios

La aplicación permite que múltiples usuarios se comuniquen entre sí a través de una red local o internet. El servidor maneja múltiples conexiones de clientes, retransmitiendo los mensajes que recibe a todos los demás clientes conectados.

3. Interfaz de usuario

La interfaz de usuario se implementa utilizando la biblioteca tkinter. Los usuarios pueden ingresar su nombre y mensajes en campos de entrada y ver los mensajes recibidos en tiempo real en un área de texto. La función pedir_nombre() solicita el nombre del usuario al inicio.

4. Conexión entre clientes y un servidor

Los sockets se utilizan para establecer la conexión entre los clientes y el servidor. El servidor escucha en un puerto específico (50123) y acepta conexiones entrantes. Cuando un cliente se conecta, se agrega a un diccionario clientes_conectados que mantiene un registro de todos los clientes conectados.

5. Manejo de errores

El código maneja adecuadamente los errores de conexión y desconexión de clientes. Si un cliente se desconecta inesperadamente, se maneja en la función broadcast(), eliminando al cliente del registro y notificando a los demás.

6. Comandos especiales

El cliente implementa comandos especiales:

/listar: Envía una solicitud al servidor para recibir la lista de usuarios conectados.

/quitar: Permite al usuario desconectarse del servidor enviando el mensaje /quitar, que cierra la conexión de manera segura.

Explicación del funcionamiento de la comunicación a nivel de código entre el cliente y el servidor a través del uso de sockets

Se importaron estas librerías para la realización del chat

socket: Esta librería es esencial para la creación de sockets en Python, lo que permite la comunicación entre el cliente y el servidor.
threading: Se utiliza para manejar múltiples conexiones de clientes simultáneamente en el servidor.
tkinter: Se utiliza para la creación de la interfaz gráfica del cliente, aunque no es parte de la comunicación por sockets.

Clientes:

```
import socket
import threading
import tkinter as tk
from tkinter import scrolledtext, messagebox
```

Servidor

```
import socket  # Imp
import threading  #
```

Puertos Utilizados
Se utiliza el puerto **50123** tanto en el cliente como en el servidor para establecer la comunicación.
Se importan las librerías necesarias y se configuran la dirección IP y el puerto donde el servidor escuchará las conexiones entrantes.

Análisis código del Servidor

Se crea un socket que permite conexiones TCP.
Se asocia el socket con la dirección IP y el puerto.
Se inicia la escucha de conexiones entrantes.

```
import socket  # Importa el módulo 'socket' para la comunicación p
import threading  # Importa el módulo 'threading' para trabajar co

# Diccionario para mantener un registro de los clientes conectados
clientes_conectados = {}

# Semáforo para sincronizar el acceso a los clientes conectados
sem = threading.Semaphore()

# Configuración del servidor
host = '127.0.0.1'  # Define la dirección IP en la que el servidor
puerto = 50123  # Define el número de puerto en el que el servidor

# Crea un socket de tipo AF_INET (IPv4) y SOCK_STREAM (TCP).
socketServidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Asocia el socket con la dirección y el puerto especificados.
socketServidor.bind((host, puerto))

# Comienza a escuchar conexiones entrantes.
socketServidor.listen()
```

Función broadcast(origen, mensaje)
La función broadcast retransmite los mensajes a todos los clientes conectados, excepto al origen. Esto lo hace recorriendo el diccionario de clientes conectado y enviando el mensaje a cada cliente menos al origen
Función manejar_cliente(socketConexion, nombre_cliente):
Esta función maneja la conexión y comunicación de un cliente específico. Aquí se reciben los mensajes del clientes ya sea comando como son listar o quitar , o mensajes para los demás clientes usando broadcast ,también hace uso del manejo de excepción

Con while true hacemos que acepte conexiones de nuevos clientes y crea un hilo para manejarlos. Llamando a manejar_cliente en un nuevo hilo para permitir múltiples conexiones simultáneas.

Análisis del Código del Cliente

Función recibir_mensajes(socket)
Recibe y muestra mensajes del servidor decodificándolo e insertándolo en la interfaz gráfica. También maneja la pérdida de conexión mostrando un mensaje de error
Función enviar_mensaje(event=None)
Envía mensajes ingresados por el usuario al servidor, acá usamos evento el cual nos permite llamar a la función cuando se presiona enter o cuando el usuario toca enviar ,si el mensaje no está vacío, lo envía al servidor, también configuramos los distintos envíos de comandos al servidor
Función cerrar_conexion()
Cierra la conexión del cliente y la ventana de la aplicación pasando el parámetro /quitar lo cual le dice al servidor que termine el bucle y cierre la conexión
Función iniciar interfaz ()
Se encargar de iniciar la interfaz gráfica tkinter para nuestro chat

Código fuente y captura de funcionamiento tanto del servidor como del cliente

cliente

```
import socket
import threading
import tkinter as tk
from tkinter import scrolledtext, messagebox
# Función para recibir mensajes del servidor
def recibir_mensajes(socket):
    while True:
        try:
            mensaje = socket.recv(1024).decode()
            if not mensaje:
                break
            texto_chat.config(state=tk.NORMAL)
            texto_chat.insert(tk.END, mensaje + '\n')
            texto_chat.config(state=tk.DISABLED)
            texto_chat.yview(tk.END)
        except ConnectionError:
            messagebox.showerror("Error", "Se ha perdido la conexión con el servidor.")
            break
# Función para enviar mensajes al servidor
def enviar_mensaje(event=None):
    mensaje = entrada_mensaje.get()
    if mensaje:
        texto_chat.config(state=tk.NORMAL)
        texto_chat.insert(tk.END, f"Tú: {mensaje}\n")
```

```
        texto_chat.config(state=tk.DISABLED)
        texto_chat.yview(tk.END)
        if mensaje == '/quitar':
            cliente.send(mensaje.encode())
            ventana_chat.quit()
        elif mensaje == '/listar':
            cliente.send(mensaje.encode())
        else:
            cliente.send(mensaje.encode())
        entrada_mensaje.delete(0, tk.END)
# Función para cerrar la conexión y la ventana
def cerrar_conexion():
    cliente.send('/quitar'.encode())
    cliente.close()
    ventana_chat.quit()
# Configuración del cliente
host = '127.0.0.1'
puerto = 50123

cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cliente.connect((host, puerto))
# Función para iniciar la interfaz gráfica
def iniciar_interfaz():
    global ventana_chat, entrada_mensaje, texto_chat
    ventana_chat = tk.Tk()
    ventana_chat.title("Chat Cliente")
```

```
    texto_chat = scrolledtext.ScrolledText(ventana_chat, wrap=tk.WORD, state=tk.DISABLED, width=50, height=20)
    texto_chat.grid(column=0, row=0, padx=10, pady=10, columnspan=2)
    entrada_mensaje = tk.Entry(ventana_chat, width=40)
    entrada_mensaje.grid(column=0, row=1, padx=10, pady=10)
    entrada_mensaje.bind("<Return>", enviar_mensaje)  # Envía mensaje al presionar Enter
    boton_enviar = tk.Button(ventana_chat, text="Enviar", width=10, command=enviar_mensaje)
    boton_enviar.grid(column=1, row=1, padx=10, pady=10)
    boton_quitar = tk.Button(ventana_chat, text="salir", width=10, command=cerrar_conexion)
    boton_quitar.grid(column=1, row=2, padx=10, pady=10)
    ventana_chat.protocol("WM_DELETE_WINDOW", cerrar_conexion)
    ventana_chat.mainloop()
# Solicita el nombre del usuario
def pedir_nombre():
    ventana_nombre = tk.Tk()
    ventana_nombre.title("Ingresa tu nombre")
    etiqueta_nombre = tk.Label(ventana_nombre, text="Ingresa tu nombre:")
    etiqueta_nombre.pack(pady=10)
    entrada_nombre = tk.Entry(ventana_nombre, width=40)  # Aumentar el tamaño
    entrada_nombre.pack(pady=10)
    def enviar_nombre(event=None):
        nombre = entrada_nombre.get()
        if nombre:
            cliente.send(nombre.encode())
            ventana_nombre.destroy()
        else:
            messagebox.showwarning("Advertencia", "El nombre no puede estar vacío.")
```

```
    entrada_nombre.bind("<Return>", enviar_nombre)  # Enviar con Enter
    boton_enviar_nombre = tk.Button(ventana_nombre, text="Enviar", command=enviar_nombre)
    boton_enviar_nombre.pack(pady=10)

    # Centrar la ventana
    ventana_nombre.update_idletasks()  # Actualiza el tamaño de la ventana
    x = (ventana_nombre.winfo_screenwidth() // 2) - (ventana_nombre.winfo_width() // 2)
    y = (ventana_nombre.winfo_screenheight() // 2) - (ventana_nombre.winfo_height() // 2)
    ventana_nombre.geometry(f"+{x}+{y}")  # Posicionar en el centro
    ventana_nombre.mainloop()
# Hilo para recibir mensajes
thread_recv = threading.Thread(target=recibir_mensajes, args=(cliente,))
thread_recv.daemon = True
thread_recv.start()
# Inicia el flujo del programa
pedir_nombre()
iniciar_interfaz()
cliente.close()
```

Servidor

```
import socket  # Importa el módulo 'socket' para la comunicación por socket.
import threading  # Importa el módulo 'threading' para trabajar con hilos.
# Diccionario para mantener un registro de los clientes conectados
clientes_conectados = {}
# Semáforo para sincronizar el acceso a los clientes conectados
sem = threading.Semaphore()
# Configuración del servidor
host = '127.0.0.1'  # Define la dirección IP en la que el servidor escuchará conexiones.
```

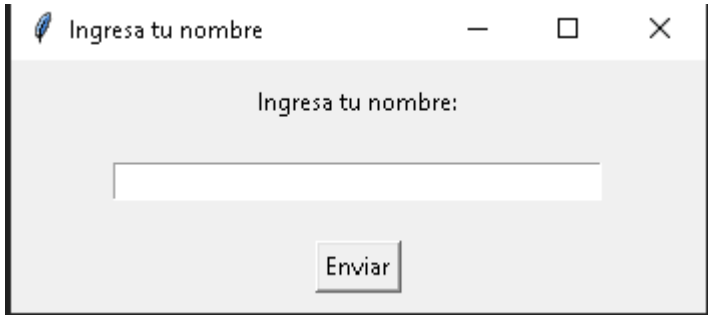
```
puerto = 50123  # Define el número de puerto en el que el servidor escuchará conexiones.
# Crea un socket de tipo AF_INET (IPv4) y SOCK_STREAM (TCP).
socketServidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Asocia el socket con la dirección y el puerto especificados.
socketServidor.bind((host, puerto))
# Comienza a escuchar conexiones entrantes.
socketServidor.listen()
# Función para enviar mensajes a todos los clientes, excepto el origen
def broadcast(origen, mensaje):
   for cliente_socket in clientes_conectados:
      if cliente_socket != origen:
         try:
            cliente_socket.send(mensaje.encode())  # Envía el mensaje codificado a cada cliente.
         except ConnectionError:
            # Manejar desconexiones inesperadas eliminando el cliente del registro.
            cliente_a_eliminar = clientes_conectados.pop(cliente_socket, None)
            if cliente_a_eliminar:
               print(f"Cliente {cliente_a_eliminar} desconectado inesperadamente.")

def manejar_cliente(socketConexion, nombre_cliente):
   try:
      # Añade el nuevo cliente a la lista de conectados
      clientes_conectados[socketConexion] = nombre_cliente

      while True:
         mensajeRecibido = socketConexion.recv(1024).decode()  # Recibe un mensaje del cliente y lo
decodifica.
```

```
            if mensajeRecibido.startswith('/listar'):
                # Si el mensaje comienza con '/listar', se genera una lista de nombres de clientes y se envía.
                lista_usuarios = "USUARIOS CONECTADOS:\n"
                for idx, nombre in enumerate(clientes_conectados.values(), 1):
                    lista_usuarios += f"usuario {idx}: {nombre}\n"
                socketConexion.send(lista_usuarios.encode())
            elif mensajeRecibido == '/quitar':
                # Si el mensaje es '/quitar', se sale del bucle, lo que permite desconectar al cliente.
                break
            else:
                # Si el mensaje no es un comando especial, se prepara el mensaje a enviar.
                mensaje_enviar = f"{nombre_cliente}: {mensajeRecibido}"
                # Usa semáforos para sincronizar la transmisión de mensajes
                sem.acquire()
                broadcast(socketConexion, mensaje_enviar)  # Llama a la función para enviar el mensaje a todos
                sem.release()
        except ConnectionResetError:
            pass
        finally:
            # Finaliza cuando el cliente se desconecta y realiza limpieza.
            print(f"Desconectado el cliente {nombre_cliente}")
            socketConexion.close()  # Cierra la conexión del cliente.
            sem.acquire()
            del clientes_conectados[socketConexion]  # Elimina al cliente del registro.
            sem.release()
    while True:
```

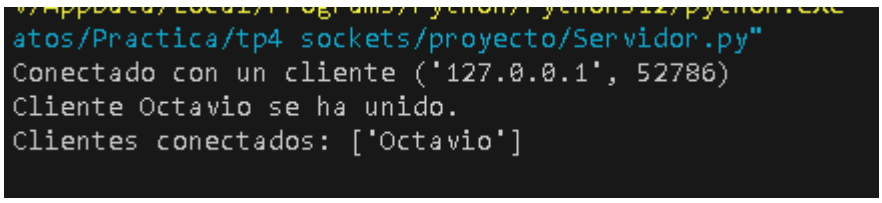
```
    socketConexion, addr = socketServidor.accept()  # Acepta una nueva conexión entrante.
    print("Conectado con un cliente", addr)  # Muestra la información de la conexión entrante.
    nombre_cliente = socketConexion.recv(1024).decode()  # Recibe el nombre del cliente que se ha unido.
    print(f"Cliente {nombre_cliente} se ha unido.")
    clientes_conectados[socketConexion] = nombre_cliente  # Agrega al cliente al registro de clientes conectados.
    # Inicia un hilo para manejar al cliente y pasa la conexión y el nombre del cliente como argumentos.
    cliente_thread = threading.Thread(target=manejar_cliente, args=(socketConexion, nombre_cliente))
    cliente_thread.start()
    # Muestra la lista de nombres de los clientes conectados.
    print("Clientes conectados:", list(clientes_conectados.values()))
```

Captura de funcionamiento

1. Primero iniciamos el servidor y luego iniciamos nuestro chat el cual nos pedirá ingresar nuestro nombre



Una vez echo podemos ver en la terminal que nos dice que usuario o cliente se conecto a nuestro servidor y a través de que puerto



2. Conectamos un segundo cliente en este caso Fernando

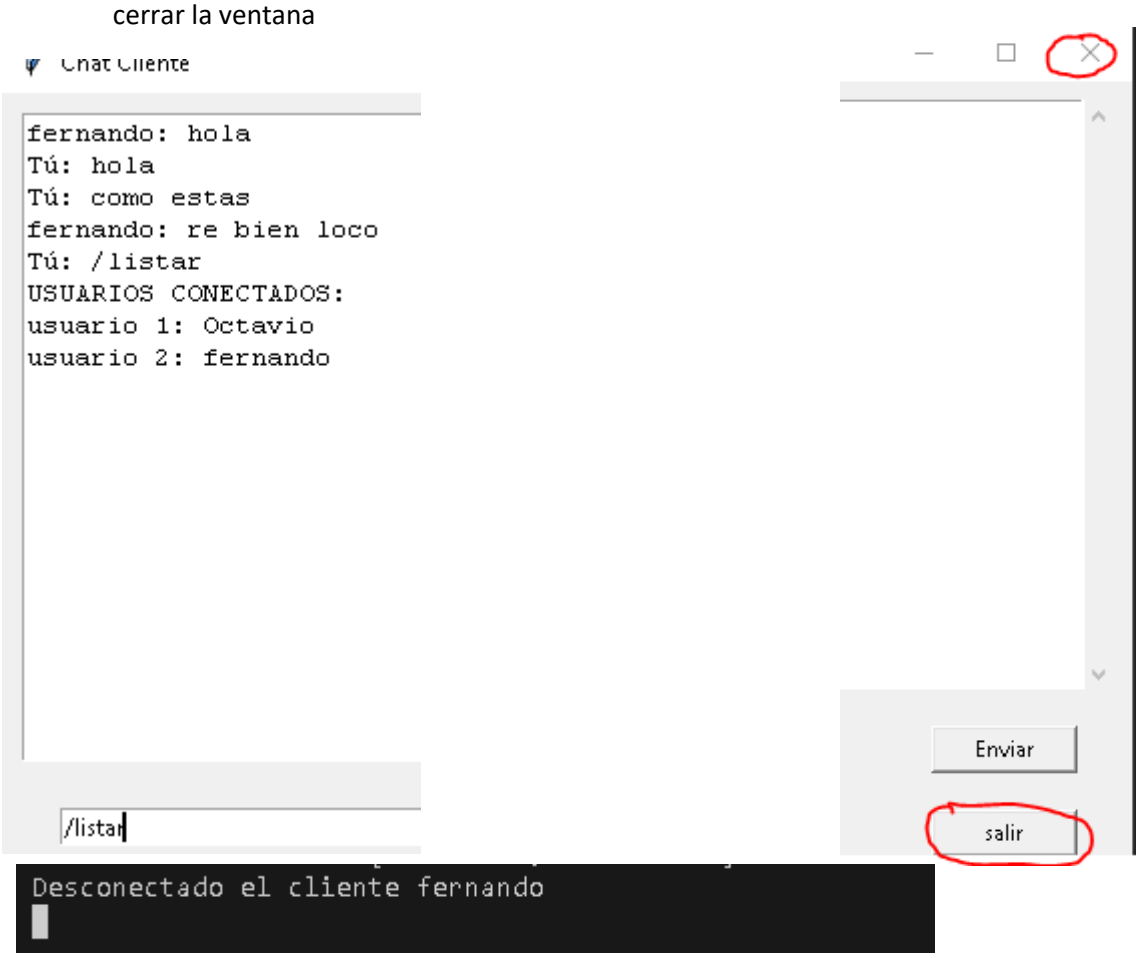
```
Conectado con un cliente ('127.0.0.1', 52786)
Cliente Octavio se ha unido.
Clientes conectados: ['Octavio']
Conectado con un cliente ('127.0.0.1', 52813)
Cliente fernando se ha unido.
Clientes conectados: ['Octavio', 'fernando']
```

Como vemos nos dice que clientes se unieron, cuales están conectados en ese momento y en que puertos

3. Procedemos a iniciar el chat entre los usuarios

Como vemos podemos comunicarnos y tener respuesta entre los clientes

4. Probamos el uso de los comandos,aca podemos usar /listar para listar los usuario y con /quitar podemos cerrar la conexión como se ve o tambien tenemos el boton salir o el de cerrar la ventana



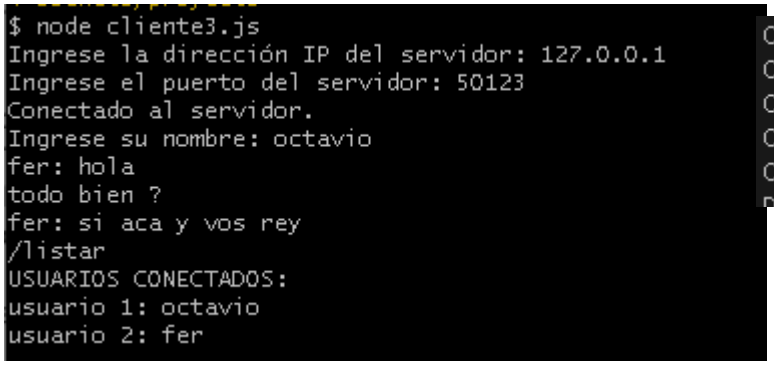
Ejercicio 2:

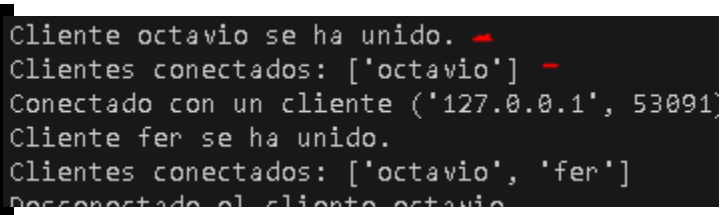
Escenario1

Utilizando como base el desarrollo del ejercicio N°1: diseñar e implementar un programa, basado en la comunicación por sockets, sobre alguno de los siguientes escenarios: 1. Nuevo Cliente de Chat

Para este nuevo cliente del chat usamos el lenguajes de JavaScript usaremos Node.js para la realización del chat

El nuevo cliente del chat solicita al usuario que ingresa parámetros como ser el host ip, el servidor y su nombre si todo salió bien se mostrará un mensaje de conexión exitosa y se procederá a iniciar





Código del cliente 3 en js:

```
const net = require('net');
const readline = require('readline');

// Crear interfaz para la entrada y salida
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

// Solicitar IP y puerto del servidor
rl.question('Ingrese la dirección IP del servidor: ', (host) => {
  rl.question('Ingrese el puerto del servidor: ', (port) => {
    const client = new net.Socket();

    // Conectar al servidor
    client.connect(port, host, () => {
      console.log('Conectado al servidor.');

      rl.question('Ingrese su nombre: ', (name) => {
        client.write(name);
        startChat(client);
      });
    });

    client.on('data', (data) => {
      console.log(data.toString());
    });

    client.on('error', (err) => {
      console.error('Error de conexión:', err.message);
    });
```

```
    client.on('close', () => {
      console.log('Conexión cerrada.');
      rl.close();
    });
  });
})
// Función para iniciar el chat
function startChat(client) {
  rl.on('line', (input) => {
    if (input === '/quitar') {
      client.write(input);
      client.destroy(); // Cierra la conexión
    } else if (input === '/listar') {
      client.write(input);
    } else {
      client.write(input);
    }
  });
}
```

Escenario 2 Estado HTTP

Se creo un programa que lee el archivo y realiza solicitudes https a través de un archivo de texto(urls.txt) que simula solicitudes REQUEST HTTP el cual contendrá las URLS a verificar
Para cada URL, el programa envía una solicitud HTTP (ya sea http o https) y obtiene el estado de la respuesta.
Si la URL es válida, el programa extrae la dirección IP del servidor y utiliza la API ip-api.com para obtener información de geolocalización sobre esa IP.
Por último se imprime el estado HTTP y la información de geolocalización (si está disponible) en la consola

```
octav@DESKTOP-UGAINOJ MINGW64 /e/lic. sistema/4to año/Redes de Datos/Practica/tp4 sockets/proyecto
$ node verificarStatus.js

Resultados de las solicitudes HTTP:
: getaddrinfo ENOTFOUND redesdedatos2024.com
------------------------------------------------------------
: 200 OKwww.google.com/
Geolocalización de -> www.google.com: {"status":"success","country":"United States","countryCode":"US","region":"CA
"lat":37.422,"lon":-122.084,"timezone":"America/Los_Angeles","isp":"Google LLC","org":"Google LLC","as":"AS15169 Go
------------------------------------------------------------
: 404 NOT FOUNDorg/status/404
Geolocalización de -> httpbin.org: {"status":"success","country":"United States","countryCode":"US","region":"VA","
38,"lon":-77.4874,"timezone":"America/New_York","isp":"Amazon.com, Inc.","org":"AWS EC2 (us-east-1)","as":"AS14618
------------------------------------------------------------
: 500 INTERNAL SERVER ERROROO
Geolocalización de -> httpbin.org: {"status":"success","country":"United States","countryCode":"US","region":"VA","
38,"lon":-77.4874,"timezone":"America/New_York","isp":"Amazon.com, Inc.","org":"AWS EC2 (us-east-1)","as":"AS14618
------------------------------------------------------------
```

Código usado para el programa

```
const fs = require('fs');//Módulo que permite interactuar con el sistema de archivos, en este caso para leer archivos.
//Módulos que permiten realizar solicitudes HTTP y HTTPS.
const http = require('http');
const https = require('https');

// Esta funcio realiza una solicitud HTTP a la URL proporcionada y devuelve el estado de la respuesta (código y mensaje).
function getHttpStatus(url) {
  return new Promise((resolve, reject) => {
    const client = url.startsWith('https') ? https : http;
    const request = client.get(url, (res) => {
      resolve(`${res.statusCode} ${res.statusMessage}`);
    });

    request.on('error', (err) => {
      reject(err.message);
    });

    // Timeout para la solicitud
    request.setTimeout(5000, () => { // 5 segundos
      request.abort(); // Aborta la solicitud
      reject('Timeout de la solicitud');
    });
  });
}

// esta Funcion obtiene información de geolocalización para una dirección IP utilizando la API ip-api.com.
async function getGeolocation(ip) {
  return new Promise((resolve, reject) => {
    http.get(`http://ip-api.com/json/${ip}`, (res) => {
      let data = '';
      res.on('data', (chunk) => {
        data += chunk;
      });
      res.on('end', () => {
        const info = JSON.parse(data);
        resolve(info);
      });
    }).on('error', (err) => {
```

```
            reject(err.message);
        });
    });
}

// Función Lee y procesa el archivo (file) que contiene URLs, realiza solicitudes HTTP para cada una
y obtiene su estado, así como la geolocalización de su IP.
async function checkUrls(file) {
    try {
        const data = fs.readFileSync(file, 'utf8');
        const urls = data.split('\n').filter(Boolean); // Filtrar líneas vacías
        const results = [];

        for (const url of urls) {
            try {
                const status = await getHttpStatus(url);
                results.push(`${url}: ${status}`);

                // Agregar geolocalización
                const ipMatch = url.match(/\/\/([^\/]+)/);
                if (ipMatch && ipMatch[1]) {
                    const geolocation = await getGeolocation(ipMatch[1]);
                    results.push(`Geolocalización de -> ${ipMatch[1]}: ${JSON.stringify(geolocation)}`);
                }
            } catch (error) {
                results.push(`${url}: ${error}`);
            }

            // Agregar separador después de cada resultado
            results.push('------------------------------------------------------------');
        }

        console.log('\nResultados de las solicitudes HTTP:');
        results.forEach(result => console.log(result));
    } catch (error) {
        console.error('Error al leer el archivo:', error);
    }
}

// Ejecutar el chequeo de URLs
checkUrls('urls.txt');//urls.txt contiene las urls a verificar
```