Instituto Tecnológico de Buenos Aires

Trabajo Práctico Especial 2025/1

Protocolos de Comunicación
72.07

Grupo 16

Integrantes:

Nicolas Koron (64094) - nkoron@itba.edu.ar
José Maria Benegas Lynch (64242) - jobenegaslynch@itba.edu.ar
Santiago Maffeo (64245) - smaffeo@itba.edu.ar

1. Descripción de los protocolos y aplicaciones

1.1 Protocolo Socks5

Se diseñó y desarrolló un protocolo aplicado a un servidor que atiende a conexiones. El mismo es una versión de SOCKS5 (RFC 1928) que cuenta con modo autenticado y modo no autenticado.

1.1.1 Autenticación

Al iniciar una conexión al proxy socks, el cliente inicia una etapa de negociación en la que se define el método a utilizar. Nuestra implementación cuenta con autorización a partir de usuario y contraseña, según lo definido en el RFC 1929, y también cuenta con la posibilidad de conectarse sin autenticación (La activación de esta última se puede manejar desde la aplicación cliente).

1.1.2 Métricas de uso

El servidor cuenta con una interfaz destinada a la recolección de métricas volátiles para el monitoreo del uso del proxy. Esta funcionalidad permite observar tanto estadísticas globales del sistema como información detallada sobre cada conexión individual de los usuarios.

Las estadísticas globales incluyen los siguientes campos:

- **Total de conexiones**: cantidad acumulada de conexiones atendidas desde el inicio del servidor.
- **Conexiones actuales**: número de conexiones activas en un momento dado.
- **Bytes enviados desde el cliente al servidor remoto**: cantidad total de datos transmitidos en esa dirección a través del proxy.
- **Bytes enviados desde el servidor remoto al cliente**: cantidad total de datos recibidos por el cliente a través del proxy.
- **Errores de envío**: cantidad de fallos ocurridos al intentar enviar datos.
- **Errores de recepción**: cantidad de fallos ocurridos al intentar recibir datos.
- **Errores de resolución DNS**: fallos registrados durante la resolución de nombres de dominio.
- **Errores del servidor**: errores internos detectados durante el funcionamiento del sistema.

- **Entradas no soportadas**: cantidad de comandos o datos recibidos en formatos no reconocidos o inválidos.

- **Conexiones con resolución DNS**: cantidad de conexiones que requirieron resolución de nombres de dominio.

- **Conexiones IPv4**: cantidad de conexiones realizadas utilizando el protocolo IPv4.

- **Conexiones IPv6**: cantidad de conexiones realizadas utilizando el protocolo IPv6.

- **Errores de autenticación**: cantidad de intentos fallidos de inicio de sesión por parte de los usuarios.

Esta estructura de métricas permite un análisis preciso del estado operativo del servidor, identificando patrones de uso, fallos recurrentes y facilitando el mantenimiento y la toma de decisiones administrativas.

Además de las estadísticas globales del servidor, se almacena información detallada sobre cada conexión realizada por los usuarios autenticados. Para ello, se mantiene una estructura de árbol binario balanceado (Red-Black Tree) por usuario, donde cada nodo representa una conexión individual y contiene los siguientes campos:

- **Fecha y hora de acceso**: momento en que se estableció la conexión.

- **Dirección IP de origen**: dirección IP desde la cual se conectó el cliente.

- **Dirección IP de destino**: dirección IP del servidor remoto al que se accedió.

- **Nombre de destino**: nombre del host o dominio correspondiente al servidor remoto.

- **Puerto de origen**: puerto desde el cual se originó la conexión.

- **Puerto de destino**: puerto del servidor remoto al cual se estableció la conexión.

- **Bytes enviados**: cantidad total de datos transmitidos desde el cliente hacia el servidor remoto.

- **Bytes recibidos**: cantidad total de datos transmitidos desde el servidor remoto hacia el cliente.

- **Estado final de la conexión**: tipo de error, siguiendo la numeración estipulada por el RFC 1928, si es que ocurrió alguno durante la sesión, o bien una indicación de finalización exitosa (marcada como 0).

Cabe destacar que las conexiones no autenticadas se agrupan bajo un usuario reservado con el nombre **"Anonymous"**, a fin de mantener un registro consistente y facilitar el análisis de tráfico no identificado.

1.2 Protocolo de Comunicación

En el marco del desarrollo del sistema cliente-servidor, se diseñó e implementó un protocolo de comunicación binario propio que permite gestionar la autenticación de usuarios, la consulta de métricas y la ejecución de comandos administrativos de manera segura y eficiente. El protocolo es simple, estructurado y extensible, y está optimizado para intercambios rápidos sobre una conexión TCP persistente.

1.2.1 Autenticación

La fase inicial de toda comunicación consiste en un proceso de autenticación. El cliente envía un mensaje que contiene su nombre de usuario y contraseña, precedido por campos de versión y longitud.

En primer lugar, el cliente le enviará al servidor:

VER	RSV	ULEN	USERNAME	PLEN	PASSWORD
BYTE	BYTE	BYTE	BYTES	BYTE	BYTES

Donde:
- VERSION (1 byte): Versión del protocolo.
- RSV (1 byte): Campo reservado para uso futuro (debe enviarse como 0x00).
- ULEN (1 byte): Longitud del campo USERNAME.
- USERNAME: Nombre de usuario (ULEN bytes).
- PLEN (1 byte): Longitud del campo PASSWORD.
- PASSWORD: Contraseña del usuario (PLEN bytes).

Cabe aclarar que Username y Password son arreglo de Bytes, de Ulen y Plen bytes respectivamente.

Luego, el servidor le responde al cliente:

VER	RSV	STATUS	ROLE
BYTE	BYTE	BYTE	BYTES

Donde:
- STATUS (1 byte): 0x00 si la autenticación fue exitosa, 0x01 si se trata de un error general del servidor y un 0x02 si se trata de un *bad request* de parte del usuario.
- ROLE (1 byte): 0x00 si el usuario es regular (USER), 0x01 si tiene permisos administrativos (ADMIN).

1.2.2 Flujo para usuarios regulares

Una vez ingresado como un usuario regular (no administrador) y recibir la respuesta del servidor sobre la correcta autenticación, se recibe una respuesta adicional que confirma el estado del pedido,

El Servidor le debe mandar al Cliente usuario:

VER	RSV	STATUS	BODYLEN	BODY
BYTE	BYTE	BYTE	4 BYTES	N BYTES

Donde:
- STATUS (1 byte): 0x00 si no hubo errores, 0x01 si se trata de un error general del servidor y un 0x02 si se trata de un *bad request* de parte del usuario.
- BODYLEN (4 bytes): Longitud del campo BODY, codificada en orden de bytes de red (*big-endian*).
- BODY (N bytes): Texto plano con las métricas del usuario autenticado. Este campo puede incluir información como número de conexiones, bytes transferidos, tiempo total conectado, entre otros.

1.2.3 Flujo para administradores

Si el usuario autenticado tiene rol ADMIN, el cliente está habilitado a enviar comandos administrativos.

El Cliente (administrador) le envía al servidor:

VER	RSV	CMD	ULEN	USERNAME
BYTE	BYTE	BYTE	BYTE	BYTES

Donde:
- CMD (1 byte): 0x00 para STATS, 0x01 para CONFIG.
- ULEN: Longitud del USERNAME de destino (puede ser 0).
- USERNAME: Usuario objetivo (opcional para STATS).

Si se envía CMD en 0x00 y en ULEN algo distinto a cero, el protocolo entiende que se están pidiendo las estadísticas de un usuario en específico. En cambio, si se envía ULEN, con 0x00, el protocolo asume que el administrador está pidiendo las métricas globales del servidor.

Luego el Servidor le responde al Cliente:

VER	RSV	STATUS
BYTE	BYTE	BYTE

Donde:
- STATUS: 0x00 si se acepta el comando, 0x01 si se trata de un error general del servidor y un 0x02 si se trata de un *bad request* de parte del usuario.
- En caso de STATS, se envían métricas en texto plano
- En caso de CONFIG, el servidor espera comandos adicionales.

1.2.4 Comandos de configuración de administrador

Si el administrador entro en modo configuración entonces:

Primero el Cliente le debe mandar al Servidor:

VER	RSV	CODE	ULEN	USERNAME	PLEN	PASSWORD	BUFFERSIZE
BYTE	BYTE	BYTE	BYTE	BYTES	BYTE	BYTES	BYTE

Donde:
- CODE (1 byte): Código del comando específico:
    - 0x00: Cambiar tamaño del buffer (usa BUFFERSIZE).
    - 0x01: Habilitar autenticación sin usuario.
    - 0x02: Deshabilitar autenticación sin usuario.
    - 0x03: Agregar usuario (requiere USERNAME y PASSWORD).
    - 0x04: Eliminar usuario (requiere USERNAME).
    - 0x05: Promover usuario a administrador (requiere USERNAME).

Y por último, el servidor le debe responder al Cliente:

VER	RSV	STATUS
BYTE	BYTE	BYTE

Donde:
- STATUS: 0x00 si el comando de métricas fue exitoso, 0xFF si el comando de configuración fue exitoso, 0x01 si hubo un error general del servidor y un 0x02 si se trata de un *bad request* de parte del usuario.

1.2.5 Decisiones tomadas sobre el protocolo cliente

Durante el diseño del protocolo de comunicación cliente-servidor, se tomaron una serie de decisiones orientadas a lograr un sistema eficiente, extensible y consistente con estándares existentes. A continuación se detallan las principales elecciones de diseño:

- **Formato binario**: se optó por un protocolo binario, tanto para comandos como para respuestas. Esta decisión se tomó por coherencia con SOCKS5 —protocolo base del sistema— y permite minimizar el tamaño de los mensajes, acelerar su procesamiento y reducir ambigüedades en el parsing.

- **Transporte sobre TCP**: el protocolo se diseñó para funcionar sobre una conexión **TCP persistente**, aprovechando su confiabilidad y control de flujo. Esto permite que las comunicaciones se realicen en orden, sin pérdidas, y facilita el manejo de estados en la aplicación.

- **Sin handshake inicial**: a diferencia de SOCKS5, el protocolo comienza directamente con la autenticación, omitiendo una fase de negociación de métodos. Esta simplificación reduce la latencia y evita un paso innecesario dado que el

método de autenticación está definido por configuración del servidor.

- **Un comando a la vez (modelo sin interacción continua)**: se definió que el protocolo sea **sincrónico y no interactivo**, en el sentido de que cada mensaje del cliente corresponde a un único comando. El servidor responde y finaliza el ciclo, sin mantener sesiones interactivas prolongadas.

- **Respuesta inmediata tras autenticación**: si el usuario autenticado es regular (no administrador), el servidor responde directamente con sus métricas personales tras un login exitoso. Esta decisión busca **reducir pasos innecesarios** y brindar feedback inmediato sin requerir comandos adicionales por parte del cliente.

- **Códigos de estado homogéneos**: todas las respuestas del servidor incluyen un campo STATUS con valores estándar:

    - 0x00: operación exitosa.
    - 0x01: error interno del servidor.
    - 0x02: solicitud mal formada (bad request).

  Esta estructura uniforme facilita el tratamiento de errores en cualquier fase del protocolo.

- **Big-endian para campos multibyte**: los campos de más de un byte (por ejemplo, BUFFERSIZE) se codifican en **orden big-endian**, en línea con las convenciones de red (network byte order).

- **Roles y control de acceso**: la autenticación devuelve el rol (ROLE) del usuario, lo que permite al cliente diferenciar entre usuarios comunes y administradores, asignando permisos adecuados.

- **Timeouts en la conexión**: para evitar sesiones colgadas o recursos ocupados indefinidamente, se implementaron **timeouts** tanto en la fase de autenticación como en la espera de comandos. Si el cliente no responde en un tiempo razonable, la conexión es cerrada automáticamente.

- **Métricas en texto plano**: si bien el protocolo es binario, las métricas se devuelven en texto plano para facilitar el debugging, el análisis por humanos y el registro en logs sin necesidad de decodificación adicional.

- **Soporte a futuro**: se reservó un campo de versión (VERSION) y un campo reservado (RSV) en los mensajes de autenticación, permitiendo compatibilidad hacia futuras extensiones del protocolo.

1.3 Aplicación Cliente

La aplicación cliente permite conectarse al servidor, autenticarse y, según los privilegios, consultar métricas o ejecutar comandos administrativos. El programa está escrito en C y utiliza sockets TCP bloqueantes, un multiplexor basado en selector, y una máquina de estados para organizar el flujo de la comunicación.

1.3.2 Decisiones tomadas sobre la aplicación cliente

Durante el desarrollo de la aplicación cliente, se tomaron múltiples decisiones de diseño y arquitectura con el objetivo de lograr un cliente robusto, extensible y eficiente para interactuar con el servidor. A continuación, se detallan las principales:

- **Uso de sockets TCP bloqueantes:** dado que el cliente está diseñado para manejar una única conexión a la vez y seguir un flujo definido de interacción, se optó por utilizar sockets en modo bloqueante. Esto simplifica el control del flujo y evita la complejidad de manejar múltiples estados o eventos asincrónicos.

- **Modelo de programación secuencial:** la lógica del cliente sigue una estructura paso a paso, en la que cada etapa (autenticación, envío de comando, recepción de respuesta) se ejecuta en orden y de forma clara. Esto favorece la legibilidad del código y facilita el debugging.

- **Tamaño de buffer configurable**: el cliente utiliza un buffer de tamaño predefinido (32 KiB), suficientemente amplio para manejar mensajes administrativos y métricas. Además, se permite la modificación del tamaño del buffer desde el cliente administrador.

- **Timeouts de espera**: se configuró un timeout de 5 segundos para el selector, de manera que se eviten bloqueos indefinidos si el servidor no responde.

- **Separación clara de responsabilidades**: el código cliente está modularizado en archivos que manejan tareas específicas:

    - clientAuth: manejo de la autenticación.

    - clientRequest: envío de comandos administrativos o de métricas.

    - clientConfig: manejo de las respuestas a comandos de configuración.

    - tcpClientUtil: creación y conexión del socket cliente.

    - handleStatsRead: recepción y parsing de métricas.

- **Protocolo binario estructurado**: el cliente construye y parsea paquetes binarios según el protocolo definido (con campos como VERSION, RSV, STATUS, ROLE,

etc.), respetando orden big-endian en los valores multibyte.

- **Validaciones estrictas de respuesta**: se realiza verificación de versión, byte reservado y códigos de estado en todas las respuestas del servidor. Esto evita comportamientos indefinidos frente a datos corruptos o mal formados.

- **Mensajes de feedback claros**: ante errores o eventos importantes (por ejemplo, autenticación fallida o éxito en un comando), el cliente imprime mensajes descriptivos para facilitar la interacción con el usuario y el debugging.

1.3.3 Comandos de la aplicación cliente

Una vez ejecutado el servidor proxy, es posible modificar su configuración en tiempo de ejecución mediante el protocolo binario de administración implementado. Esto se realiza utilizando la aplicación cliente, la cual permite conectarse, autenticarse (si la autenticación está habilitada) y enviar comandos administrativos.

Las siguientes opciones de configuración están disponibles para usuarios con rol de administrador. Todos los comandos, dado que requieren privilegios de administrador, se deben correr con el -l para loguearse como administrador. La única excepción es el de métricas del propio usuario, que también debe ser con el -l, pero no requiere de privilegios de administrador:

Agregar un nuevo usuario: Permite incorporar un usuario con contraseña al sistema. Flag: -u
Eliminar un usuario existente: Elimina a un usuario del sistema. Flag: -r
Promover usuario a administrador: Otorga privilegios administrativos a un usuario existente. Flag: -m
Modificar el tamaño del buffer: Cambia dinámicamente el tamaño del buffer utilizado por el servidor. Flag: -b
Habilitar autenticación opcional: Permite que los clientes se conecten sin autenticarse. Flag: -n
Deshabilitar autenticación opcional: Fuerza a todos los clientes a autenticarse con usuario y contraseña. Flag: -N
Conexión a puerto: especifica el puerto de conexión del servidor. Flag -p
Conexión a dirección (address): Define la dirección IP o el nombre del host del servidor al cual debe conectarse el cliente. Flag -a

Además se pueden consultar métricas del sistema:

Métricas propias del usuario logueado: Para obtener las métricas propias del usuario logueado, simplemente se debe invocar a la aplicación cliente con el flag de -l y sus credenciales, sin ninguna flag adicional
Métricas globales del sistema: Solo accesibles a administradores. Incluyen el total de conexiones, bytes transferidos, usuarios activos, entre otros. También muestra todas las conexiones con sus detalles del servidor. Flag: -G
Métricas de un usuario específico: También sólo accesible a administradores. Flag: -s

2. Problemas encontrados durante el desarrollo

Durante el desarrollo del trabajo práctico se presentaron diversos desafíos técnicos y organizativos que requirieron múltiples iteraciones, análisis y pruebas para ser superados.

1. Manejo del tiempo y la planificación
Uno de los principales obstáculos fue la gestión adecuada del tiempo. Al tratarse de un proyecto largo con múltiples componentes, cliente, servidor, protocolo, interfaz de administración, etc., fue complejo estimar correctamente el esfuerzo requerido para cada parte. En particular, dado a que éramos tres integrantes tuvimos que manejar los tiempos para también poder rendir lo que teníamos que rendir de exámenes finales.

2. Diseño e implementación de un protocolo desde cero
Crear un protocolo de comunicación binario propio implicó tomar decisiones importantes de diseño desde los primeros pasos. Se debió definir la estructura de los mensajes, el orden de los campos, los valores posibles para cada comando y estado, y garantizar que el parsing de cada mensaje fuera robusto y extensible. Esta tarea, si bien enriquecedora, también introdujo errores sutiles difíciles de detectar, como malinterpretaciones de longitud o problemas con el orden de los bytes (endianness). Además, se tuvo que tener en cuenta que lo que planteamos en la teoría, no necesariamente se podía implementar en la práctica, por lo que tuvimos que ir cambiando sobre la marcha el protocolo, hasta llegar a uno totalmente implementable

3. Complejidad en el código: fugas de memoria
En el desarrollo del servidor y del cliente se evidenciaron problemas de gestión de memoria, especialmente vinculados al uso de estructuras dinámicas como buffers, listas o máquinas de estados. A lo largo del desarrollo se detectaron **fugas de memoria** que no se liberaban correctamente en rutas de error o al finalizar conexiones.

3. Limitaciones

A pesar de cumplir con los objetivos funcionales del sistema, existen diversas limitaciones en la implementación actual que restringen su escalabilidad, flexibilidad y capacidad de administración:

- **Cantidad máxima de usuarios registrados**: el sistema permite registrar como máximo **10 usuarios**. Este valor está definido en tiempo de compilación, lo cual impide adaptar dinámicamente la capacidad del sistema según la carga o el entorno de despliegue. Este valor contempla tanto administradores como usuarios sin privilegios.

- **Falta de gestión de roles**: los usuarios se crean siempre con un rol fijo (usuario estándar) y **no es posible modificar su estado** posteriormente. En particular, **no se permite remover privilegios de administrador,** en tiempo de ejecución, solamente se pueden otorgar privilegios de administrador a un usuario ya creado.

- **Límite de conexiones concurrentes**: el servidor soporta hasta **500 usuarios conectados simultáneamente**. Este número puede no ser suficiente en entornos de alta concurrencia o en pruebas de carga intensiva.

- **Persistencia limitada**: la lista de usuarios registrados no persiste entre reinicios del servidor. Si se apaga o reinicia el sistema, todos los registros de usuario se pierden, lo que restringe su uso en entornos productivos.

4. Conclusiones

El desarrollo del presente proyecto permitió aplicar e integrar múltiples conceptos vistos en la cursada de Protocolos de Comunicación, especialmente en lo que refiere al diseño e implementación de aplicaciones cliente-servidor utilizando sockets no bloqueantes y protocolos personalizados.

Uno de los mayores logros fue la implementación de un servidor proxy funcional compatible con SOCKSv5 ([RFC1928]) con soporte de autenticación por usuario y contraseña ([RFC1929]). El sistema es capaz de manejar múltiples conexiones concurrentes, administrar usuarios en tiempo de ejecución y recolectar métricas de monitoreo, todo a través de un protocolo binario propio de administración, extensible y bien estructurado.

Otro aspecto destacable fue el diseño e implementación de un protocolo binario propio para comunicación entre el cliente de monitoreo y el servidor. Este protocolo se diseñó desde cero, con campos bien definidos y una semántica clara.

Asimismo, se logró desarrollar una aplicación cliente que permite interactuar con el servidor proxy para monitoreo y configuración. Esta herramienta facilita la supervisión del estado del sistema y la administración remota del mismo sin necesidad de reiniciar el servidor, cumpliendo con uno de los requerimientos más desafiantes del enunciado.

Durante el proceso se fortalecieron habilidades relacionadas al diseño de protocolos de aplicación, manejo de errores en comunicaciones asincrónicas, y gestión eficiente de recursos.

En definitiva, el trabajo práctico no solo cumplió con los objetivos propuestos, sino que también abrió la puerta a posibles mejoras y extensiones que podrían convertir al sistema en una solución aún más completa y robusta para entornos reales.

5. Posibles Extensiones

A pesar de lograr implementar correctamente el servidor, la aplicación cliente y el protocolo para la comunicación entre ellos, se pueden considerar posibles extensiones:

- **Persistencia de métricas y usuarios**: Actualmente, las métricas y la configuración de usuarios son volátiles y se pierden ante un reinicio del servidor. Una posible mejora sería incorporar persistencia mediante archivos o bases de datos ligeras (como SQLite), permitiendo recuperar el estado anterior tras reinicios o fallas.
- **Cifrado de la comunicación cliente-servidor**: Si bien el protocolo binario implementado funciona correctamente, actualmente no cifra los mensajes. Integrar TLS permitiría garantizar confidencialidad e integridad de los datos, especialmente en ambientes productivos o con redes no confiables.
- **Comandos interactivos y una interfaz gráfica para el cliente de monitoreo**: Una interfaz visual facilitaría la interacción de administradores con el sistema, permitiendo visualizar métricas, gestionar usuarios y cambiar configuraciones de forma más amigable.

6. Ejemplos de Prueba

Para validar la correcta operación y performance del proxy SOCKS5, se realizaron pruebas utilizando un script Python que permitió simular un escenario de carga controlada y medir métricas relevantes.
Inicialmente se intentó usar JMeter, sin embargo, por dificultades técnicas derivadas del entorno (WSL y no Linux nativo), la ejecución a través del proxy SOCKS5 no fue fiable ni reproducible. Por ello se optó por un script Python que ejecuta múltiples requests concurrentes con curl usando autenticación SOCKS5.
Las llamadas curl durante las pruebas se realizaron a distintos endpoints distribuidos en el servidor local corriendo en localhost puerto 8000 (un servidor de python que retornaba algo al hacerle curl), generando URLs dinámicas con parámetros y rutas variadas para simular un tráfico heterogéneo y evitar sesgos o bloqueos por concentración en una sola ruta.
Por último, se armó un stress test que no funciona de manera local, sino que se conecta a distintas IPs y dominios públicos. Este script, con 100 threads y un total de 3000 requests, no espera respuesta antes de lanzar la siguiente. De esta forma se alcanza una concurrencia efectiva de más de 500 usuarios.
Posteriormente, este mismo script fue corregido, ya que contenía algunas IPv4 inválidas que impactaron negativamente el porcentaje de éxito, como también cambiamos los usuarios, 20.000 conexiones con 2000 threads. Se mantuvieron ambos resultados en el informe: uno como evidencia de alta concurrencia, y otro con mayor confiabilidad.
Todos los códigos se pueden encontrar en el anexo.

6.1 Test de Carga

Para evaluar la capacidad del proxy SOCKS5 bajo condiciones de carga, se realizaron 500 solicitudes concurrentes simuladas a través de un script Python que utiliza curl con autenticación SOCKS5. El test empleó hasta 50 conexiones simultáneas estimadas, distribuyendo las solicitudes en distintos endpoints para evitar sesgos o posibles bloqueos por DoS.

El resultado mostró que todas las solicitudes (500) fueron exitosas, sin errores de conexión ni autenticación. Esto indica que el proxy soporta una carga moderada con concurrencia considerable manteniendo estabilidad y sin rechazos ni caídas. (ver imágenes en sección 6.7)

Los datos del servidor proxy confirman 500 conexiones totales gestionadas, sin errores de envío, recepción, autenticación ni resolución DNS, validando la capacidad del proxy para manejar simultáneamente esta cantidad de clientes.

La prueba de stress adicional orientada a simular una carga de mayor escala, donde se lanzaron 20000 requests totales utilizando 2000 hilos concurrentes, accediendo a múltiples dominios públicos (no locales).

En este test se obtuvieron 199972 solicitudes exitosas y 28 fallidas.Las 28 solicitudes fallidas fueron por timeout, no por fallos del proxy. Esto refleja saturación por alta demanda, sin caída del servicio. Siendo que los ips que más fallaban por timeout eran de amazon, microsoft y google.

6.2 Tiempo de Respuesta

El tiempo promedio de respuesta en la primer prueba fue de 0.111 segundos por solicitud, con baja latencia y sin picos irregulares.

En el stress test final de 20000 requests, el tiempo promedio fue 0.004589 segundos, mostrando un excelente comportamiento incluso bajo alta concurrencia, atribuible a la alta eficiencia del proxy al manejar solicitudes externas resueltas previamente.

6.3 Throughput de bytes

- En la prueba de 500 requests, se transfirieron 5.000.000 bytes en ~2.8 segundos, lo que representa un throughput de ~1.779.635 bytes/segundo.

- En el stress test de 20000 requests, se transfirieron 14.949.303 bytes en 91.78 segundos, lo que da un throughput de ~162.890 bytes/segundo.

Este valor más bajo refleja el overhead natural en conexiones externas, mayor resolución DNS, y latencia de red. Sin embargo, sigue siendo un resultado robusto considerando la escala y la dispersión de destinos.

6.4 Usuarios concurrentes

Adicionalmente, se ejecutó un script de python donde se mantienen 501 conexiones concurrentes para asegurarnos cumplir con este requisito. Efectivamente, el proxy no registró errores y funcionó correctamente para este caso.

6.5 Fiabilidad de los resultados

Los resultados obtenidos son consistentes entre el cliente y el servidor proxy, sin errores reportados en ningún punto del flujo de datos o en la autenticación. Esto asegura la fiabilidad de las métricas obtenidas.

El proxy gestionó exitosamente más de 500 conexiones y su respectivo tráfico sin mostrar signos de saturación o errores, validando la escalabilidad y estabilidad bajo las condiciones testadas.

Para un análisis más profundo sobre la degradación del throughput y la capacidad máxima de conexiones simultáneas, se recomienda realizar pruebas incrementales aumentando progresivamente la carga y monitoreando recursos del sistema (CPU, memoria, red).

El enfoque utilizado, basado en múltiples endpoints y concurrencia distribuida, evita sesgos típicos de ataques DoS y aporta datos representativos sobre el comportamiento real del proxy en producción.

6.6 Uso de proxy en Firefox

Se configuró el navegador Firefox para correr con nuestro Proxy. Se visitaron varias páginas web al mismo tiempo, incluyendo distintas pestañas de youtube, y no se encontraron problemas.

6.7 Imágenes de los Test

A continuación se detallan los resultados de los distintos tests:

Imagen primer stress test:

```
====== RESUMEN DEL TEST ======
Total requests: 500
Requests exitosos: 500
Requests fallidos: 0
Tiempo promedio de respuesta: 0.111 segundos
Throughput total (bytes/seg): 1779635.34
Bytes totales recibidos: 5000000
Duración total del test: 2.810 segundos
Máximo de conexiones simultáneas (estimado): 50
```

```
==== GLOBAL METRICS ====
total_connections: 500
current_connections: 0
bytes_client_to_remote: 87500
bytes_remote_to_client: 10141000
dns_resolutions_connections: 500
send_errors: 0
receive_errors: 0
dns_resolution_errors: 0
ipv4_connections: 0
ipv6_connections: 0
server_errors: 0
auth_errors: 0
unsupported_input: 0
host_unreachable: 0
```

Imagen segundo stress test con ips inválidas:

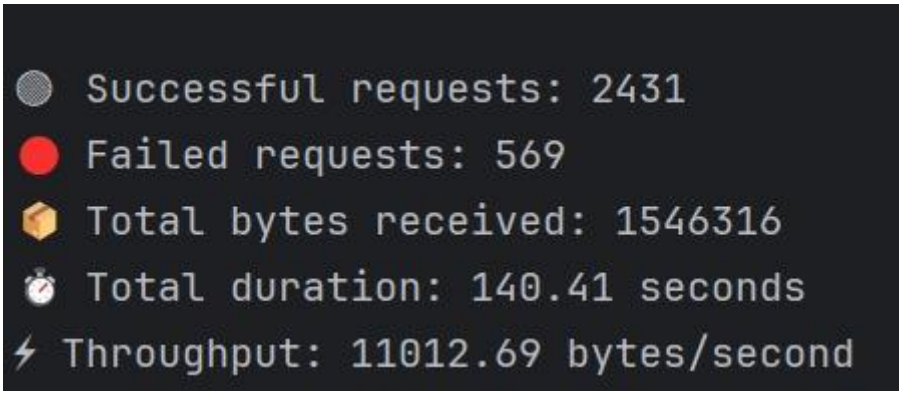
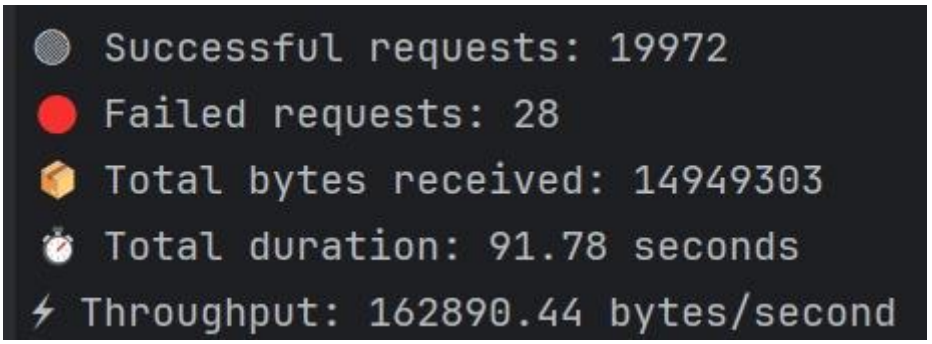


Imagen segundo stress test con ips válidas:



6.8 Conclusiones

Las pruebas realizadas con el proxy SOCKS5 han demostrado un comportamiento sólido y confiable bajo las condiciones de carga evaluadas. A continuación se resumen las principales conclusiones:

- **Performance**
  El proxy demostró una latencia muy baja y una alta capacidad de procesamiento, incluso bajo estrés. Los errores por timeout en pruebas masivas no comprometen la estabilidad ni representan fallos del servidor. El diseño eficiente de interpretación de mensajes y reenvío de datos resulta en una muy buena experiencia final.
- **Escalabilidad**
  Se observaron buenos resultados hasta 2000 hilos concurrentes, con una tasa de éxito superior al 99%. El proxy evita sobrecarga en memoria, mantiene eficiencia y responde correctamente incluso con alta cantidad de conexiones paralelas.
- **Disponibilidad y estabilidad**
  No se reportaron errores de conexión, caídas o corrupción de datos. La arquitectura es resistente y tolera escenarios de carga sostenida sin colapsar.
- **Discrepancia de bytes**
  Se registró una diferencia entre los bytes medidos por el script (contenido de respuesta) y los reportados por el proxy (total en tránsito).
  Esto es esperado y no implica pérdida: el proxy contabiliza headers, reintentos y metadatos, mientras que el script mide solo el payload útil.
- **Fiabilidad de resultados:**
  La consistencia entre las métricas reportadas desde el cliente y el servidor valida la confiabilidad de las pruebas realizadas. No se observaron discrepancias significativas en la cantidad de conexiones ni en el comportamiento general del proxy.

- **Pruebas de stress:**
  Si bien estas primeras pruebas confirman la capacidad del proxy para manejar 500 usuarios concurrentes, es necesario profundizar mediante pruebas incrementales que permitan determinar:

  ¿Cuál es la máxima cantidad de conexiones simultáneas que soporta el proxy?
  Probamos con más de 500 usuarios concurrentes dado que el máximo que manejamos según lo estipulado en el código es de 501, y anduvo correctamente.

  ¿Cómo se degrada el throughput al aumentar la carga?
  El throughput del proxy muestra una degradación gradual conforme se incrementa la cantidad de conexiones simultáneas. A bajas cargas (50-100 conexiones), el rendimiento se mantiene muy estable, cercano al máximo throughput medido (~2.1 MB/s).

  Sin embargo, al aumentar la concurrencia a 500 y 1000 conexiones, se observa una caída del throughput del 16% y 24.5% respectivamente, indicando que el proxy empieza a sentir los efectos de la carga y que la eficiencia de transferencia se reduce conforme la demanda crece. Estos resultados evidencian una buena escalabilidad inicial, con degradación progresiva que debe ser monitoreada en escenarios de producción para asegurar la calidad del servicio. Cabe aclarar que como manejamos hasta 500 conexiones simultáneas, el dato del de 1000 conexiones se ve afectado por esto (ya que si las conexiones concurrentes es > 500, esperamos a que se vayan liberando).

  Para el segundo stress test se observó que el throughput del proxy mejora inicialmente a medida que se incrementa la cantidad de solicitudes concurrentes (pasando de ~198 KB/s con 1000 requests a ~359 KB/s con 10.000).
  Sin embargo, al superar ese umbral, el rendimiento comienza a decaer significativamente. En el test de 20.000 solicitudes, el throughput cae a ~162 KB/s, lo que representa una reducción del 54% respecto al pico máximo observado.
  Esto sugiere que, aunque el proxy escala bien hasta cierto punto, eventualmente entra en una zona de saturación, probablemente por limitaciones de CPU, resolución DNS o manejo de sockets.
  Se recomienda seguir probando con granularidad intermedia (entre 10.000 y 20.000 conexiones) y monitorear recursos del sistema para identificar los cuellos de botella y ajustar parámetros del servidor.

7. Guia de instalacion

7.1 Requisitos previos

Antes de compilar e instalar la aplicación, asegúrese de contar con:

- Un sistema operativo tipo UNIX (probado en Linux).
- GCC con una versión compatible con C11.
- Herramienta de desarrollo, **make**
- Permisos suficientes para ejecutar y compilar binarios locales.

7.2 Instalación

El proyecto puede instalarse de dos maneras:

- **Clonando el repositorio** desde su fuente remota mediante *git*.
- **Descargando el archivo comprimido (ZIP)** y descomprimiendolo en el directorio deseado.

Ambas opciones permiten obtener el código fuente completo listo para compilar y ejecutar.

7.3 Compilación

Ejecutar el siguiente comando para compilar todas las aplicaciones:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ make clean all
```

Esto compilará tanto el servidor proxy como el cliente de monitoreo/configuración.

7.4 Archivos generados

Una vez completada la compilación, se generarán los siguientes ejecutables:

- Server: ejecutable del servidor SOCKSv5 con monitoreo.
- Client: cliente de monitoreo/configuración administrativa.

Ambos estarán en el directorio bin ubicado en la raíz.

7.5 Inicialización de aplicaciones

El servidor del proxy se puede correr con:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/server [ARGS]
```

La aplicación cliente se debe correr con el comando:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client <command> [ARGS]
```

7.6 Verificación de la instalación

Al hacer ./bin/server -h, se debería poder visualizar la sección de ayuda del servidor:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/server -h
Usage: ./bin/server [OPTION]...

   -h               Imprime la ayuda y termina.
   -l <SOCKS addr>  Dirección donde servirá el proxy SOCKS.
   -L <conf  addr>  Dirección donde servirá el servicio de management.
   -p <SOCKS port>  Puerto entrante conexiones SOCKS.
   -P <conf port>   Puerto entrante conexiones configuracion
   -u <name>:<pass> Usuario y contraseña de usuario que puede usar el proxy. Hasta 10.
   -v               Imprime información sobre la versión versión y termina.
```

También se puede hacer ./bin/server -v y obtener la versión y del servidor:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/server -v
socks5v version 1.0
ITBA Protocolos de Comunicación 2025/1 -- Grupo 16
```

Similarmente, se puede hacer ./bin/cliente -h y también se obtendrá la sección de ayuda del servidor:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -h
Usage: ./bin/client [options]
  -b, --buffer-size <n>
  -n, --no-auth
  -N, --no-no-auth
  -u, --add-user <user:pass>
  -r, --remove-user <user>
  -m, --make-admin <user>
  -l, --login <user:pass>   (requerido)
  -p, --port <port>
  -a, --address <addr>
  -G, --global-metrics
  -s, --specific-metrics <user>
  -h, --help
```

8. Instrucción para la configuración

Lo único necesario para configurar el servidor es correr el comando de inicialización con el flag de -a que agrega un administrador al servidor para poder conectarse y manejar correctamente la aplicación cliente:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/server -a user:pass
```

9. Ejemplos de configuración y monitoreo

9.1 Agregar y remover usuarios

Para agregar un usuario se debe enviar el <usuario:contrasena> como argumento después del flag ‘-u’:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -u santi:maff
## Authentication successful for Admin role
#Ok, user added!
```

Para remover el usuario, se debe enviar <usuario> dado que no aceptamos nombres de usuario repetidos, con el flag ‘-r’:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -r santi
## Authentication successful for Admin role
#Ok, user removed!
```

Para poder agregar un administrador, se debe hacer como el remover usuario, pero con el flag de ‘-m’:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -u santi:maff
## Authentication successful for Admin role
#Ok, user added!
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -m santi
## Authentication successful for Admin role
#Ok, user is now admin!
```

9.2 Métricas y monitoreo

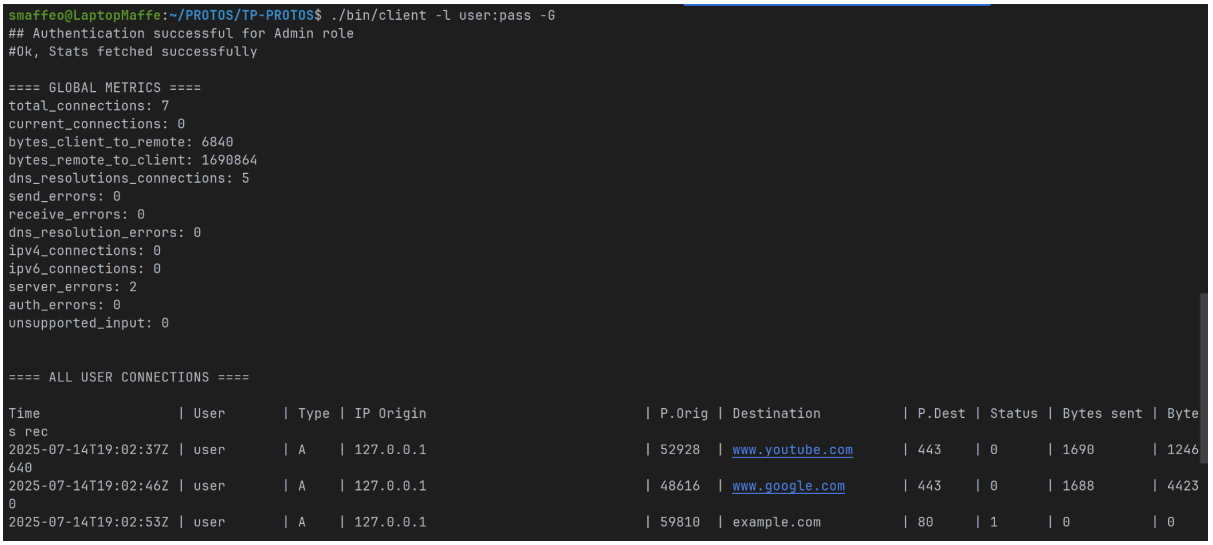
9.2.1 Conexiones propias del usuario

Para ver conexiones propias, se debe conectarse al servidor y loguearse:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass
## Authentication successful for Admin role
#Ok, Stats fetched successfully

Time                 | User       | Type | IP Origin                       | P.Orig | Destination          | P.Dest | Status | Bytes sent | Byte
s rec
2025-07-14T19:02:37Z | user       | A    | 127.0.0.1                       | 52928  | www.youtube.com      | 443    | 0      | 1690       | 1246
640
2025-07-14T19:02:46Z | user       | A    | 127.0.0.1                       | 48616  | www.google.com       | 443    | 0      | 1688       | 4423
0
2025-07-14T19:02:53Z | user       | A    | 127.0.0.1                       | 59810  | example.com          | 80     | 1      | 0          | 0

2025-07-14T19:03:05Z | user       | A    | 127.0.0.1                       | 40192  | www.example.com      | 80     | 1      | 0          | 0

2025-07-14T19:03:42Z | user       | A    | 127.0.0.1                       | 35872  | canyouseeme.org      | 80     | 0      | 158        | 956

smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$
```

9.2.2 Métricas globales del servidor

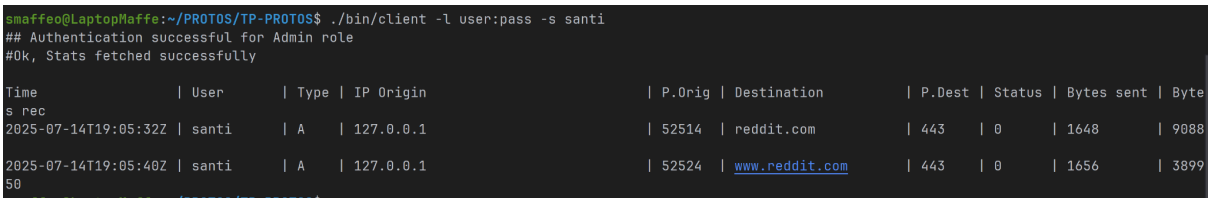
Para ver las estadísticas globales del servidor y todas las conexiones, un administrador se debe conectar y utilizar la flag ‘-G’



```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -G
## Authentication successful for Admin role
#Ok, Stats fetched successfully

==== GLOBAL METRICS ====
total_connections: 7
current_connections: 0
bytes_client_to_remote: 6840
bytes_remote_to_client: 1690864
dns_resolutions_connections: 5
send_errors: 0
receive_errors: 0
dns_resolution_errors: 0
ipv4_connections: 0
ipv6_connections: 0
server_errors: 2
auth_errors: 0
unsupported_input: 0


==== ALL USER CONNECTIONS ====

Time                 | User       | Type | IP Origin                       | P.Orig | Destination          | P.Dest | Status | Bytes sent | Byte
s rec
2025-07-14T19:02:37Z | user       | A    | 127.0.0.1                       | 52928  | www.youtube.com      | 443    | 0      | 1690       | 1246
640
2025-07-14T19:02:46Z | user       | A    | 127.0.0.1                       | 48616  | www.google.com       | 443    | 0      | 1688       | 4423
0
2025-07-14T19:02:53Z | user       | A    | 127.0.0.1                       | 59810  | example.com          | 80     | 1      | 0          | 0
```

9.2.3 Conexiones de un usuario

Para ver las conexiones de un usuario en específico, el administrador debe conectarse y utilizar la flag ‘-s’ seguido por el nombre del usuario cuyas estadísticas quiere visualizar:

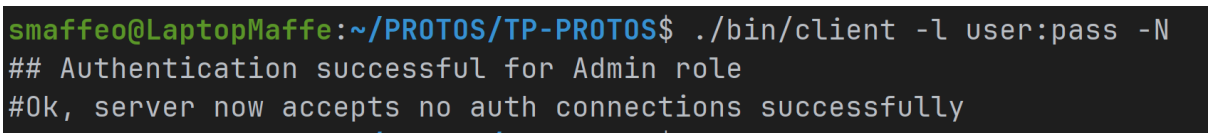


```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -s santi
## Authentication successful for Admin role
#Ok, Stats fetched successfully

Time                 | User       | Type | IP Origin                       | P.Orig | Destination          | P.Dest | Status | Bytes sent | Byte
s rec
2025-07-14T19:05:32Z | santi      | A    | 127.0.0.1                       | 52514  | reddit.com           | 443    | 0      | 1648       | 9088

2025-07-14T19:05:40Z | santi      | A    | 127.0.0.1                       | 52524  | www.reddit.com       | 443    | 0      | 1656       | 3899
50
```

9.3 Método de autenticación del servidor

9.3.1 Aceptar conexiones anónimas

Para cambiar la configuración del servidor y aceptar conexiones anónimas (es decir sin autenticación), un administrador debe conectarse y utilizar el flag ‘-n’:



```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -N
## Authentication successful for Admin role
#Ok, server now accepts no auth connections successfully
```

9.3.2 No aceptar conexiones anónimas

Para cambiar la configuración del servidor y no aceptar conexiones anónimas (es decir con autenticación), un administrador debe conectarse y utilizar el flag ‘-N’:

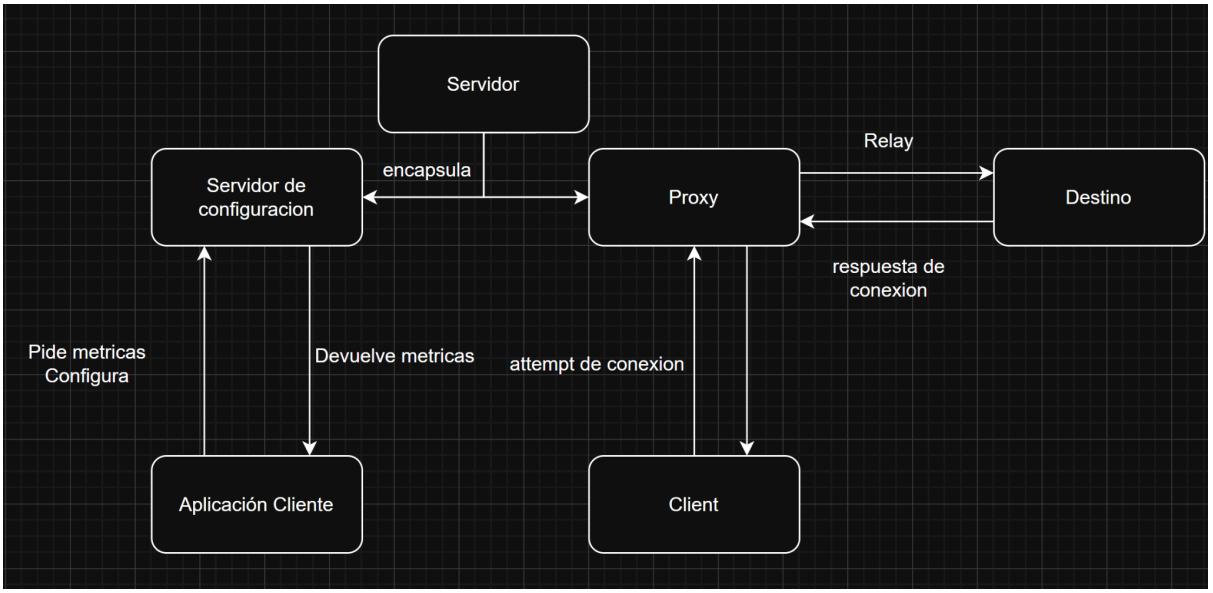
9.4 Buffer size

Para cambiar el tamaño del buffer, el administrador debe conectarse y utilizando la flag ‘-b’, agregar el tamaño de buffer deseado:

```
smaffeo@LaptopMaffe:~/PROTOS/TP-PROTOS$ ./bin/client -l user:pass -b 2000
## Authentication successful for Admin role
#Ok, buffer size changed successfully
```

10. Documento de diseño del proyecto

A continuación se puede ver un diseño que refleja el flujo del sistema al igual que protocolos se usan:



11. Anexo

En el siguiente espacio dejamos los códigos utilizados en python para probar el proxy y ejecutar el test de stress:

Código para ejecutar la prueba de stress completa (500 usuarios y 50 en simultáneo)

```
import concurrent.futures
import subprocess
import random
import time

PROXY = "user:pass@127.0.0.1:1080"
BASE_URL = "http://localhost:8000"
NUM_USERS = 500
MAX_WORKERS = 50  # Hilos concurrentes (conexiones simultáneas estimadas)

def generate_endpoint(i):
    paths = [
        "/",
        f"/resource/{i % 50}",
        f"/api/data?id={i}",
        f"/info?user={i}",
        f"/status/{random.randint(1,100)}"
    ]
    return BASE_URL + random.choice(paths)

def run_curl(url):
    cmd = [
        "curl",
        "--silent",
        "--socks5-hostname", PROXY,
        url
    ]
    start = time.time()
    try:
        result = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, timeout=15)
        duration = time.time() - start
        out_bytes = len(result.stdout)
        return {
            "url": url,
            "returncode": result.returncode,
            "duration": duration,
            "bytes": out_bytes,
            "err": result.stderr.decode().strip()
        }
    except subprocess.TimeoutExpired:
        duration = time.time() - start
        return {
            "url": url,
            "returncode": -1,
            "duration": duration,
            "bytes": 0,
            "err": "Timeout"
        }

def main():
    print(f"Running test with {NUM_USERS} requests and up to {MAX_WORKERS} concurrent
connections...\n")

    results = []
    start_total = time.time()
    with concurrent.futures.ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
        futures = []
        for i in range(NUM_USERS):
```

```
            url = generate_endpoint(i)
            futures.append(executor.submit(run_curl, url))

        active_connections = 0
        max_active_connections = 0

        for future in concurrent.futures.as_completed(futures):
            active_connections += 1
            max_active_connections = max(max_active_connections, active_connections)

            res = future.result()
            results.append(res)

            active_connections -= 1

            status = "OK" if res["returncode"] == 0 else "FAIL"
            print(f"[{status}] {res['url']} - {res['duration']:.3f}s - {res['bytes']} bytes - ERR: {res['err']}")

    end_total = time.time()
    total_time = end_total - start_total

    # Métricas
    total_requests = len(results)
    success_requests = sum(1 for r in results if r["returncode"] == 0)
    fail_requests = total_requests - success_requests
    avg_response_time = sum(r["duration"] for r in results) / total_requests if total_requests else 0
    total_bytes = sum(r["bytes"] for r in results)
    throughput_bytes_per_sec = total_bytes / total_time if total_time > 0 else 0

    print("\n====== RESUMEN DEL TEST ======")
    print(f"Total requests: {total_requests}")
    print(f"Requests exitosos: {success_requests}")
    print(f"Requests fallidos: {fail_requests}")
    print(f"Tiempo promedio de respuesta: {avg_response_time:.3f} segundos")
    print(f"Throughput total (bytes/seg): {throughput_bytes_per_sec:.2f}")
    print(f"Bytes totales recibidos: {total_bytes}")
    print(f"Duración total del test: {total_time:.3f} segundos")
    print(f"Máximo de conexiones simultáneas (estimado): {MAX_WORKERS}")

if __name__ == "__main__":
    main()
```

Código para ejecutar la prueba de usuarios concurrentes (Modificar el max num)

```
import concurrent.futures
import subprocess
import random

PROXY = "user:pass@127.0.0.1:1080"
BASE_URL = "http://localhost:8000"
NUM_USERS = 600

def generate_endpoint(i):
    paths = [
        "/",
        f"/resource/{i % 50}",
        f"/api/data?id={i}",
        f"/info?user={i}",
        f"/status/{random.randint(1,100)}"
    ]
    return BASE_URL + random.choice(paths)

def run_curl(url):
    cmd = [
        "curl",
        "--silent",
        "--socks5-hostname", PROXY,
        url
    ]
    try:
        result = subprocess.run(cmd, stdout=subprocess.PIPE, stderr=subprocess.PIPE, timeout=10)
        return (url, result.returncode, len(result.stdout), result.stderr.decode().strip())
    except subprocess.TimeoutExpired:
        return (url, -1, 0, "Timeout")

def main():
    with concurrent.futures.ThreadPoolExecutor(max_workers=NUM_USERS) as executor:
        futures = []
        for i in range(NUM_USERS):
            url = generate_endpoint(i)
            futures.append(executor.submit(run_curl, url))

        success = 0
        fail = 0

        for future in concurrent.futures.as_completed(futures):
            url, retcode, bytes_received, err = future.result()
            if retcode == 0:
                success += 1
            else:
                fail += 1
            print(f"[{retcode}] {url} bytes: {bytes_received} ERR: {err}")

        print(f"\nTotal usuarios concurrentes: {NUM_USERS}")
        print(f"Requests exitosos: {success}")
        print(f"Requests fallidos: {fail}")

if __name__ == "__main__":
    main()
```

Servidor de python

```
from http.server import BaseHTTPRequestHandler, HTTPServer

class SimpleHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        body = b'A' * 10_000  # 10 KB de datos
        self.send_response(200)
        self.send_header('Content-Length', str(len(body)))
        self.send_header('Content-Type', 'text/plain')
        self.end_headers()
        self.wfile.write(body)

if __name__ == '__main__':
    server = HTTPServer(('localhost', 8000), SimpleHandler)
    print("Serving on http://localhost:8000")
    server.serve_forever()
```

Stress test no local con ips invalidas:

```
import socks
import socket
import threading
import time
import random

# Configuración del proxy SOCKS5
PROXY_HOST = "127.0.0.1"
PROXY_PORT = 1080

# Destinos (IPv4 y dominios)
TARGETS = [
    ("google.com", 80),
    ("amazon.com", 80),
    ("youtube.com", 80),
    ("microsoft.com", 80),
    ("openai.com", 80),
    ("cloudflare.com", 80),
    ("142.250.64.78", 80),
    ("13.35.122.89", 80),
    ("172.217.164.110", 80),
    ("20.112.52.29", 80),
    ("104.18.12.123", 80),
    ("104.22.61.85", 80),
]

# Parámetros de carga
TOTAL_REQUESTS = 3000
NUM_THREADS = 100
```

```
REQUESTS_PER_THREAD = TOTAL_REQUESTS // NUM_THREADS
SOCK_TIMEOUT = 10

# Contadores globales
success_count = 0
failure_count = 0
total_bytes_received = 0
lock = threading.Lock()

def stress_worker(thread_id):
    global success_count, failure_count, total_bytes_received

    for _ in range(REQUESTS_PER_THREAD):
        host, port = random.choice(TARGETS)
        try:
            s = socks.socksocket(socket.AF_INET)
            s.set_proxy(socks.SOCKS5, PROXY_HOST, PROXY_PORT)
            s.settimeout(SOCK_TIMEOUT)

            s.connect((host, port))
            s.sendall(b"GET / HTTP/1.0\r\nHost: " + host.encode() + b"\r\n\r\n")

            # Leer todo lo que se pueda (hasta que corte o supere timeout)
            bytes_recv = 0
            while True:
                try:
                    chunk = s.recv(4096)
                    if not chunk:
                        break
                    bytes_recv += len(chunk)
                except socket.timeout:
                    break

            s.close()

            with lock:
                success_count += 1
                total_bytes_received += bytes_recv
        except Exception as e:
            with lock:
                failure_count += 1
            print(f"[Thread {thread_id}] Failed to connect to {host}:{port} - {e}")

threads = []

start = time.time()

for i in range(NUM_THREADS):
    t = threading.Thread(target=stress_worker, args=(i,))
    t.start()
    threads.append(t)

for t in threads:
    t.join()
```

```
duration = time.time() - start
throughput_bytes_per_sec = total_bytes_received / duration if duration > 0 else 0

print(f"\n🟢 Successful requests: {success_count}")
print(f"🔴 Failed requests: {failure_count}")
print(f"📦 Total bytes received: {total_bytes_received}")
print(f"⏱ Total duration: {duration:.2f} seconds")
print(f"⚡ Throughput: {throughput_bytes_per_sec:.2f} bytes/second")
```

Stress test no local con todas ips válidas

```
import socks
import socket
import threading
import time
import random

# Configuración del proxy SOCKS5
PROXY_HOST = "127.0.0.1"
PROXY_PORT = 1080

# Destinos (IPv4 y dominios)
TARGETS = [
    ("google.com", 80),
    ("amazon.com", 80),
    ("youtube.com", 80),
    ("microsoft.com", 80),
    ("openai.com", 80),
    ("cloudflare.com", 80),
    ("142.250.64.78", 80),
    ("172.217.164.110", 80),
    ("104.18.12.123", 80),
    ("104.22.61.85", 80),
]

# Parámetros de carga
TOTAL_REQUESTS = 20000
NUM_THREADS = 2000
REQUESTS_PER_THREAD = TOTAL_REQUESTS // NUM_THREADS
SOCK_TIMEOUT = 5

# Contadores globales
success_count = 0
failure_count = 0
total_bytes_received = 0
lock = threading.Lock()
```

```
def stress_worker(thread_id):
    global success_count, failure_count, total_bytes_received

    for _ in range(REQUESTS_PER_THREAD):
        host, port = random.choice(TARGETS)
        try:
            s = socks.socksocket(socket.AF_INET)
            s.set_proxy(socks.SOCKS5, PROXY_HOST, PROXY_PORT)
            s.settimeout(SOCK_TIMEOUT)

            s.connect((host, port))
            s.sendall(b"GET / HTTP/1.0\r\nHost: " + host.encode() + b"\r\n\r\n")

            # Leer todo lo que se pueda (hasta que corte o supere timeout)
            bytes_recv = 0
            while True:
                try:
                    chunk = s.recv(4096)
                    if not chunk:
                        break
                    bytes_recv += len(chunk)
                except socket.timeout:
                    break

            s.close()

            with lock:
                success_count += 1
                total_bytes_received += bytes_recv
        except Exception as e:
            with lock:
                failure_count += 1
            print(f"[Thread {thread_id}] Failed to connect to {host}:{port} - {e}")

threads = []

start = time.time()

for i in range(NUM_THREADS):
    t = threading.Thread(target=stress_worker, args=(i,))
    t.start()
    threads.append(t)

for t in threads:
    t.join()

duration = time.time() - start
throughput_bytes_per_sec = total_bytes_received / duration if duration > 0 else 0

print(f"\n🟢 Successful requests: {success_count}")
print(f"🔴 Failed requests: {failure_count}")
print(f"📦 Total bytes received: {total_bytes_received}")
print(f"⏱️ Total duration: {duration:.2f} seconds")
print(f"⚡ Throughput: {throughput_bytes_per_sec:.2f} bytes/second")
```