Trabajo Práctico Especial

Protocolos de Comunicación

Instituto Tecnológico de Buenos Aires

Grupo 10

Nombre	Apellido	Legajo	E-mail
Emilio Jose	Mitchell	64.590	emitchell@itba.edu.ar
Isabel	Conde	65.779	isconde@itba.edu.ar
Tomas	Varettoni	64.200	tvarettoni@itba.edu.ar
Bruno	Taccone	64.888	btaccone@itba.edu.ar

Fecha de Entrega: 11-12-2025

Docentes:

Kulesz, Sebastian

Garberoglio, Marcelo Fabio

2025

ITBA

Indice

1. Introducción

El objetivo de este informe es documentar el desarrollo integral de un servidor proxy SOCKSv5. El proyecto abarca no solo la implementación del servidor para manejar múltiples conexiones de forma eficiente, sino también la creación de una herramienta de administración para monitorearlo y configurarlo.

A lo largo del documento se describe la arquitectura elegida, se analizan las decisiones de diseño tomadas frente a los desafíos del desarrollo y se detallan los pasos necesarios para su instalación, configuración y prueba.

2. Descripción de lo desarrollado

Esta sección profundiza en la arquitectura técnica de la solución entregada. El sistema se compone de dos aplicaciones principales: un servidor proxy de alta concurrencia y un cliente de administración. La comunicación entre estas partes y con el mundo exterior se rige tanto por estándares de la industria (SOCKSv5) como por un protocolo de diseño propio para tareas de gestión.

A continuación, se describen las especificaciones de los protocolos soportados y la arquitectura interna del software desarrollado.

2.1 Protocolos

Para el funcionamiento del sistema, se ha trabajado sobre dos conjuntos de reglas de comunicación: el estándar para el proxy y uno personalizado para la gestión del servidor.

2.1.1 Protocolo SOCKSv5

El servidor desarrollado respeta estrictamente los estándares definidos en los documentos RFC 1928 (protocolo base) y RFC 1929 (autenticación). La comunicación se estructura en etapas secuenciales gestionadas por máquinas de estado, lo que garantiza que no se bloquee el flujo del servidor.

El ciclo de vida de cada conexión comienza con la fase de negociación definida en el RFC 1928. Para gestionarla de forma no bloqueante, se implementó una máquina de estados que procesa el saludo inicial del cliente.

El servidor examina los métodos ofrecidos por el cliente y acepta aquellos que estén habilitados en la configuración actual. Por defecto, se requiere autenticación por usuario y contraseña (0x02), pero es posible configurar dinámicamente el servidor para aceptar conexiones sin autenticación (0x00) mediante el protocolo de gestión. Una vez acordado el método, se da paso a la negociación bajo el RFC 1929, donde la lectura de credenciales se realiza byte a byte. De esta manera, el parser puede reconstruir dinámicamente el nombre de usuario y la contraseña en memoria. Una vez que todo fue leído, se validan los datos contra el registro interno; en caso de éxito, se pasa a la etapa de lectura de solicitudes, cerrando la sesión en caso contrario.

En la etapa de *request*, el servidor interpreta el comando enviado por el cliente; la implementación actual soporta exclusivamente el comando `CONNECT` (0x01) para el establecimiento de túneles TCP.

Respecto al direccionamiento, el sistema maneja IPv4, IPv6 y nombres de dominio. Para los nombres de dominio, se diseñó una solución de resolución DNS asincrónica utilizando `getaddrinfo_a`. A diferencia de una resolución estándar (que sería bloqueante), esta llamada lanza la consulta en segundo plano, permitiendo que se atienda a otros clientes mientras se espera la respuesta. Finalmente, una vez obtenida la dirección IP, la conexión con el servidor de destino se realiza mediante sockets no bloqueantes.

2.1.2 Protocolo de monitoreo y configuración

Para cumplir con el requerimiento de poder gestionar el servidor mientras el mismo está corriendo, se diseñó un protocolo de capa de aplicación específico que opera sobre TCP. Este protocolo permite consultar métricas y manejar usuarios en tiempo de ejecución.

Estructura del mensaje: el mensaje está compuesto por una primera parte de autenticación y una segunda parte de la petición en sí. En la primera parte, se envían usuario y contraseña, ambos campos de longitud variable precedidos por un byte que indica su longitud. Si estas credenciales de usuario no coinciden con las de un usuario administrador del servidor, la petición será rechazada. Una vez terminada la parte de autenticación, se sigue con la segunda parte. Esta inicia con un byte indicando el tipo de petición solicitado, seguido por una secuencia de argumentos, cada uno de longitud variables, precedido por un byte indicando su longitud. Para indicar que ya se enviaron todos los argumentos, se envía un byte en cero luego del último argumento.

Comandos soportados:

- Añadir usuario
- Remover usuario
- Cambiar contraseña de usuario
- Solicitar estadísticas
- Cambiar métodos de autenticación aceptados

2.2 Aplicaciones

El desarrollo de software se realizó en lenguaje C (estándar C11), priorizando la eficiencia y la modularidad.

2.2.1 Servidor proxy

El servidor es el núcleo del proyecto. Su diseño se basa en una arquitectura de bucle de eventos utilizando sockets no bloqueantes y multiplexación de entrada y salida. De esta manera, se logró manejar múltiples conexiones dentro de un mismo hilo de ejecución, haciendo al servidor más eficiente y escalable.

Para lograr este funcionamiento, se implementó una máquina de estados. Esto significa que el procesamiento de una conexión no es lineal, sino que avanza por etapas claramente definidas que permiten al servidor dejar una conexión en "espera activa" (por ejemplo, esperando una respuesta de red) y continuar atendiendo a otros usuarios sin bloquearse. Particularmente, el ciclo de vida de una conexión atraviesa las siguientes etapas:

- **`HELLO_READ / HELLO_WRITE`:** En estas etapas iniciales se procesa el saludo (*handshake*) del protocolo. El servidor lee los métodos de autenticación ofrecidos por el cliente (`HELLO_READ`) y, tras validarlos, envía la respuesta seleccionando el método adecuado o rechazando la conexión (`HELLO_WRITE`).
- **`AUTH_READ / AUTH_WRITE`:** Para la autenticación, el servidor consume las credenciales del usuario byte a byte (`AUTH_READ`) para validar el acceso. Posteriormente, envía el resultado de la validación al cliente (`AUTH_WRITE`), permitiendo o denegando el paso a la siguiente fase.
- **`REQUEST_READ`:** Una vez autenticado, el servidor lee la solicitud de conexión del cliente. Acá se determina si el destino es una dirección IP o un nombre de dominio.
- **`DNS_LOOKUP`:** Estado intermedio donde el servidor espera la resolución de un nombre de dominio (FQDN) a una dirección IP. Al utilizar resolución asincrónica, el servidor monitorea el estado de la consulta sin bloquear el hilo principal hasta obtener el resultado.
- **`DEST_CONNECT`:** En este estado, el servidor inicia y monitorea el establecimiento de la conexión TCP con el servidor de origen. Se aguarda a que el socket esté listo para escribir (indicando conexión exitosa) o reporte un error.
- **`REQUEST_WRITE`:** El servidor comunica al cliente el resultado final de su solicitud (éxito o tipo de error específico). Si la conexión fue exitosa, se prepara todo para la transferencia de datos.
- **`FORWARDING`:** Es el estado principal de transferencia. El servidor actúa como un intermediario transparente, copiando datos bidireccionalmente entre el cliente y el servidor de origen utilizando buffers intermedios.
- **`DONE / ERROR`:** Estados terminales encargados de cerrar los descriptores de archivo, liberar la memoria y destruir la sesión de manera ordenada, ya sea por finalización natural o por fallos en el protocolo.

Por otro lado, para almacenar la información relacionada a una conexión en particular se utilizó una estructura llamada *client_t*. En ella se encuentra la máquina de estados, los *file descriptors* asociados al cliente y al destino junto con buffers para cada uno, entre otras cosas. Para simplificar las cosas, se tomó la decisión de que tanto el cliente como el destino compartan estructura *client_t*. Esto resulta beneficioso pues ahorra espacio y tiene campos que ayudan al cerrado de la conexión.

Una vez llegado al estado de forwarding, cada vez que se activa un socket para lectura, se escribe todo en el buffer asociado y se activa el interés de escritura del otro lado de la conexión. Cada vez que un socket se activa para escritura se vacía el buffer correspondiente. La única forma en la que estos estados se "bloquean" es cuando los buffers se llenan, pero el servidor continua el funcionamiento normal hasta que se puedan vaciar.

Por último, para evitar la pérdida de datos durante el cierre de una conexión, se implementó un mecanismo de finalización propio de canales full-duplex. El proceso comienza cuando uno de los extremos (cliente o destino) envía un EOF. En ese momento, el lado correspondiente de la conexión se marca como *closed* y se procede a drenar y reenviar al otro extremo todos los datos aún pendientes en el buffer.

Luego, se espera que el extremo opuesto también envíe su EOF; cuando esto ocurre, ese lado de la conexión se marca como *closed*. Una vez que ambos sentidos han sido cerrados de forma independiente y que todos los datos pendientes han sido entregados, se completa la

transferencia final hacia el extremo que inició el cierre. Finalmente, se procede a cerrar ambos sockets, garantizando así un cierre ordenado sin pérdida de información.

2.2.2 Cliente de administración

Esta herramienta permite a los administradores configurar el servidor en tiempo real sin necesidad de detener el servicio. Su funcionamiento es sencillo: se conecta mediante TCP al puerto de gestión (8080 por default) y actúa como un traductor que toma los comandos ingresados por el usuario y los envía en el formato binario requerido por el protocolo (enviando primero la autenticación y luego la acción deseada). Para facilitar su uso, el paso de argumentos se diseñó manteniendo la misma lógica que en la aplicación del servidor, utilizando *flags* simples para ejecutar operaciones como agregar usuarios o consultar estadísticas de forma rápida.

3. Problemas encontrados

El desarrollo del servidor proxy implicó una curva de aprendizaje significativa. A continuación, se detallan los principales desafíos enfrentados en las etapas de diseño e implementación.

3.1 Diseño

Uno de los primeros obstáculos fue la comprensión profunda de las herramientas provistas por la cátedra (`selector`, `stm`, `buffer`, `parser`, `args`) y la arquitectura del *echo server* de referencia. Particularmente, nos resultó un gran desafío comprender cómo se debía combinar todo ello para lograr el resultado final. Esta comprensión previa incluso al diseño fue la mayor dificultad que encontramos, pero creemos que haberlo hecho con paciencia fue de suma utilidad para evitar problemas futuros.

Por otro lado, la definición de la estructura de datos que representa a un cliente requirió de múltiples iteraciones, mutando a medida que lo hacía nuestra comprensión del flujo completo del servidor. Naturalmente, empezamos con lo más básico y fuimos agregando distintas variables. Fue un tema de debate la decisión de tener una sola estructura por conexión o separarlo en cliente y destino; como fue mencionado anteriormente, por simplicidad se optó por la primera opción.

3.2 Implementación

Con respecto a la implementación, una de las dificultades que encontramos fue lograr correctamente que el las funciones sean no bloqueantes. En este mismo proceso, comprendimos la necesidad de un *logger*. Este, por su parte, trajo su propia complejidad y fue evolucionando con el tiempo: se empezó proponiendo un buffer de poco tamaño, luego uno que iba incrementando a la par que los logs y, por último, se debió decidir dónde y cómo hacer el select, ya que el *file descriptor* del *logger* se selecciona a la par que el de los clientes.

Por el lado del *parser*, las dificultades fueron similares a las planteadas en el inicio de diseño. Se utilizó mucho tiempo para comprenderlo, pero también se fue adaptando sobre la marcha. Un punto a destacar es que tardamos en darnos cuenta que el retorno de estos debía ser un estado de la máquina de estados.

Uno de los últimos problemas en aparecer fue en el cierre de las conexiones. Nuestra primera implementación cerraba ambos extremos de la conexión apenas recibía un fin de cualquiera de los dos lados. Al principio esto parecía funcionar bien cuando intentábamos conexiones individuales pero cuando al realizar el test de estrés vimos que muchas conexiones se cerraron abruptamente sin dar mensaje de error. La solución a esto fue aplicar un cierre full duplex.

4. Limitaciones de la aplicación

Aunque el sistema cumple con los requerimientos funcionales obligatorios establecidos para el Trabajo Práctico, existen ciertas restricciones relacionadas al diseño del proyecto. A continuación, se detallan las limitaciones conocidas de la implementación actual.

Por un lado, podemos mencionar la cobertura del protocolo. Nos enfocamos en la funcionalidad principal para navegar: el comando `CONNECT` vía TCP. Por lo tanto, no soportamos tráfico UDP ni otros comandos opcionales del estándar, como `BIND`. Además, respecto a la autenticación, se soportan los métodos de 'No Autenticación' (0x00) y 'Usuario/Contraseña' (0x02); no implementamos otros mecanismos más complejos definidos en el RFC.

Por otro lado, la aplicación de cliente está bastante limitada. Si bien se permite el manejo de los usuarios (agregando, borrando, cambiando sus datos), no está disponible la posibilidad de cambiar el nivel de acceso de los distintos usuarios (administradores o no), entre otras posibles configuraciones adicionales.

5. Posibles extensiones

Aunque el sistema actual cumple con los objetivos funcionales, hay varias áreas en las que podríamos evolucionar el proyecto. La mejora más significativa sería transformar la herramienta de monitoreo en una aplicación que permita al administrador mantener una sesión abierta para ejecutar múltiples consultas y configuraciones de forma continua, en lugar de tener que invocar el programa repetidamente por cada acción individual. Además, como fue mencionado en el apartado anterior, podrían agregarse nuevas funcionalidades a esta aplicación, permitiendo el acceso a más información (como los accesos de usuarios en rangos de tiempo, la lista de usuarios completa), la modificación de más parámetros (nivel de acceso, tamaños de buffer y también pudiendo restringir algunos accesos.

Otro punto donde podríamos extendernos es en el uso del logger. Actualmente, se loguea todo tanto a la salida estándar de la terminal donde se corre el servidor como el archivo “logs.txt” que se genera localmente en la máquina corriendo el servidor. Lo que se podría hacer es definir distintos niveles de logueo (DEBUG, INFO, etc.) para poder hacer la diferenciación entre un logueo a salida estándar y otro al archivo de logs.

Por el lado del protocolo, lo ideal sería seguir completando el estándar SOCKS v5. Hoy el servidor solo funciona con TCP, así que un gran paso sería agregar soporte para tráfico UDP. También se podrían sumar nuevas formas de autenticación, para no depender únicamente de usuario y contraseña y ofrecer más opciones de seguridad.

6. Conclusiones

El desarrollo de este TP representó un desafío técnico interesante. Si bien al principio nos costó bastante comprender la lógica del servidor, una vez superada esa barrera inicial pudimos avanzar con buen ritmo, ajustando y profundizando nuestro entendimiento a medida que progresaba la implementación.

Ver el trabajo terminado es muy gratificante. Logramos traducir la teoría en una herramienta práctica y, aunque fue difícil, cerrar el proyecto con el servidor andando y las herramientas de monitoreo respondiendo nos pone muy contentos.

Creemos que el desarrollo de este trabajo es un gran cierre para la materia: siempre desafiante, con conceptos que nos costaron entender, pero que nos dieron mucha gratificación cuando los logramos entender y aplicar. Valoramos especialmente haber podido comprender el funcionamiento interno de protocolos tan fundamentales y necesarios.

7. Ejemplos de prueba

7.1 Prueba de Browser

El primer paso realizado para probar pruebas individuales fue usar la herramienta `curl` para pegarle a IPs o dominios conocidos. Una vez pasada la primera prueba quisimos ver si el proxy soportaba requests más pesadas. Para esto configuramos *Firefox* para que salga a Internet vía nuestro proxy e intentamos entrar al *Meet* al cual estábamos conectados.

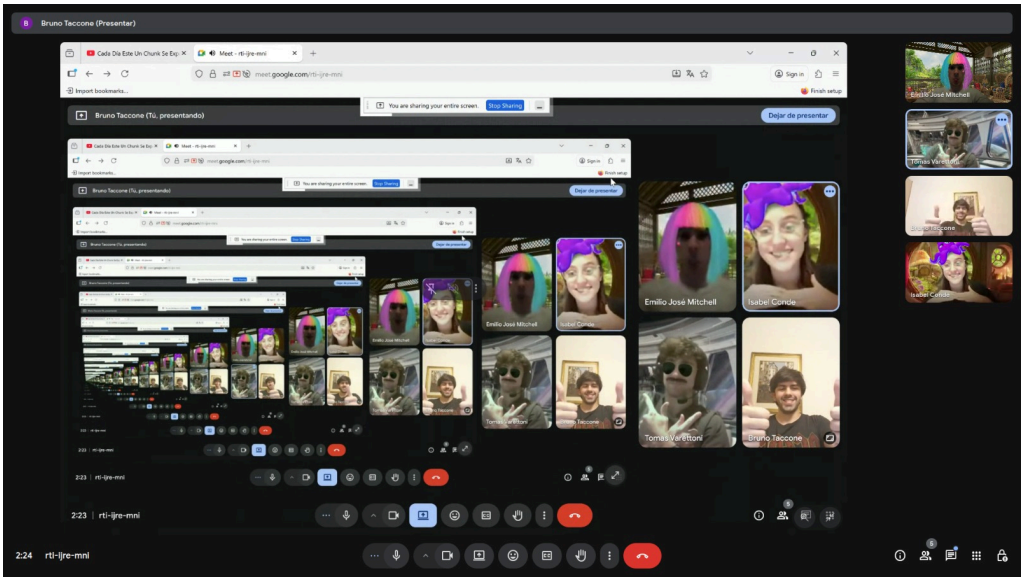

Figura 1. Captura de pantalla de un Google Meet con pantalla compartida, accedido a través del servidor proxy.

7.2 Prueba de concurrencia

El siguiente paso en los testeos fue realizar pruebas de concurrencia en el servidor. Para eso se creó el archivo *test_proxy.py* dentro de la carpeta *test*, que intenta 500 requests al mismo

dominio. Hacer este test nos ayudó a darnos cuenta de varios errores en nuestro diseño y cuellos de botella accidentales generados por los tipos de datos usados en la estructura *client_t*.

7.3 Prueba de throughput

Con el objetivo de cuantificar el impacto que introduce el servidor proxy respecto a una conexión directa, se realizó una prueba específica de *throughput* utilizando la herramienta *sockperf*. Esta herramienta permite generar un flujo sintético de tráfico TCP y medir tanto la latencia de cada mensaje como estadísticas agregadas.

La prueba consistió en hacer uso de sockperf en dos casos distintos: una conexión directa, y una conexión mediante el proxy SOCKS5 implementado. Para este último se utilizó el programa *proxychains* para poder conectar los envíos de sockperf al servidor proxy. Se utilizaron los parámetros de 600.000 mensajes por segundo y el tamaño de mensaje siendo 1024 bytes. Para ambos, el test corrió un total de 10 segundos. Para ambos, se generaron archivos tipo .csv para después interpretar los datos recopilados. Estos fueron los comandos ejecutados en linea de comando para realizar las pruebas:

```
sockperf server --tcp -i 0.0.0.0 -p 9000 // inicia el servidor sockperf

sockperf pp --stream -i 127.0.0.1 -p 9000 -m 1024 -t 10 --full-log
latency_direct.csv --histogram 1:0:200 // comando de prueba directa

proxychains4 sockperf pp --stream -i 127.0.0.1 -p 9000 -m 1024 -t 10 --full-log
latency_via_proxy.csv --histogram 1:0:200 // comando de prueba mediante proxy
```

Código 1. Comandos para llevar a cabo la prueba de throughput.

En la corrida directa, los datos que sockperf reportó llevan a que se mandaban aproximadamente 22.146 mensajes/s y se tenía un *throughput* agregado de ≈181 Mb/s con tamaño de mensaje de 1024 bytes. La latencia promedio medida fue de 22,5 µs, con una mediana de 17,4 µs y un percentil 99 de 84,1 µs.

En la corrida vía proxy, se tuvo una frecuencia de envío igual a 5606 mensajes/s y un *throughput* total de ≈46 Mb/s. La latencia promedio aumentó a 89,0 µs, con una mediana de 76,0 µs y un percentil 99 de 302,3 µs. Esto implica que, bajo la misma carga ofrecida, el servidor SOCKS5 causa que la latencia se cuadruplique y el sistema completo sólo consigue aproximadamente un 25 % del throughput en bits/s respecto al camino directo.

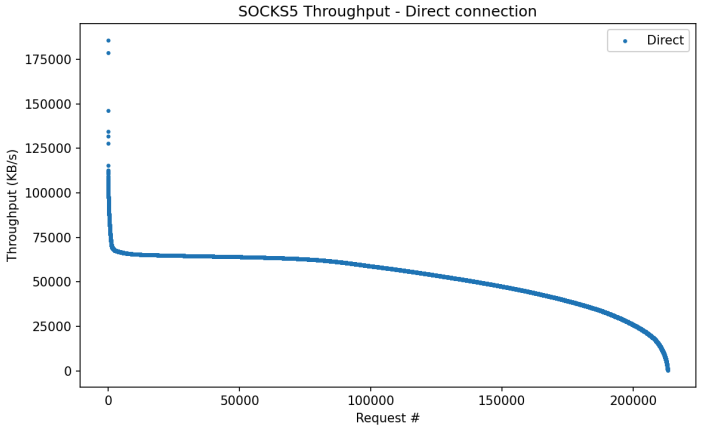

Figura 2. Throughput en KB/s sobre el número de pedido para la conexión directa.

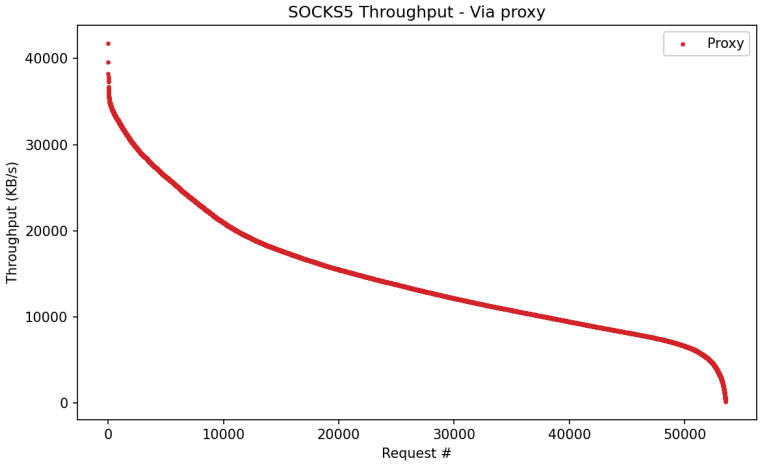

Figura 3. Throughput en KB/s sobre el número de pedido para la conexión vía proxy.

Las curvas de las Figuras 2 (conexión directa) y 3 (vía proxy) reflejan esta diferencia: la conexión directa se mantiene en valores de “throughput por paquete” mucho más altos y con una caída más suave, mientras que la curva del proxy está desplazada hacia abajo y muestra un rango más amplio de latencias. Aun así, en ambos casos las latencias se mantienen en el orden de decenas a cientos de microsegundos.

Esta prueba evidencia que el proxy se convierte en el cuello de botella cuando se le exige una tasa elevada de mensajes y expone las limitaciones generales de usar un proxy de esta manera en lugar de utilizar una conexión direct; se deben leer los pedidos y construir las respuestas, lo cual lleva más tiempo, y la utilización de un sistema de logueo del servidor proxy también aporta a esta diferencia.

7.4 Prueba de pipelining

En el contexto de este proyecto, manejar pipelining correctamente implica poder atender a varios pedidos que se le hacen al proxy SOCKS5 a través de una conexión de una manera ordenada y correcta (es decir, sin ningún tipo de pérdida de información). Para poder probar el comportamiento del servidor ante esta situación, se llevó a cabo una prueba que consiste en abrir una sola conexión hacia el servidor proxy y mandar múltiples pedidos mediante esta conexión. El mismo script que generaba la conexión después se aseguraba de que las respuestas a estos pedidos lleguen en orden y completos.

Para implementar esta prueba, se corrió el siguiente código in-line de Python que generaba un origen HTTP/1.1 con el flag de 'Connection: keep-alive' para poder mandar más de un pedido sobre una sola conexión:

```
python3 - <<'PY'
from http.server import ThreadingHTTPServer, SimpleHTTPRequestHandler
class H(SimpleHTTPRequestHandler):
    protocol_version = "HTTP/1.1"
    def end_headers(self):
        self.send_header("Connection", "keep-alive")
        super().end_headers()
ThreadingHTTPServer(("127.0.0.1", 8000), H).serve_forever()
PY
```

Código 2. Programa in-line de Python para generar un origen HTTP/1.1 con configuración keep-alive.

Después, se corría el script *test_pipelining.py* que generaba la conexión TCP hacia el servidor proxy y después verifica que las respuestas recibidas eran correctas. La respuesta recibida era la siguiente:

```
bruntacc@LAPTOP-HIRJ2COG:~/Protos/ProtosTPE$ python3 src/test/test_pipelining.py
Total responses: 20/20
Pipelining OK (responses complete and parsed in sequence)
```

Figura 4. Respuesta recibida del script test_pipelining.py.

Con esto, se puede afirmar que el servidor proxy puede manejar pipelining de manera correcta.

7.5 Prueba con distintos tamaños de buffer

Otro test de interés para evaluar el desempeño del servidor es el envío de pedidos de distintos tamaños, interpretados en el contexto del proyecto como distintos tamaños de buffer. Para probar esto, se generaron archivos de distintos tamaños con el siguiente comando:

```
dd if=/dev/zero of=64k.bin bs=1K count=64 // crea un binario de 64KB
```

Código 3. Comando para crear un binario de 64 KB

que se pedirá desde un origen HTTP inicializado por línea de comando en el mismo directorio donde se crearon los binarios (en el entorno de la prueba realizada, en el directorio */tmp*). Después, se corrieron los siguientes comandos para obtener el tiempo de respuesta del servidor proxy y de una conexión directa:

```
time curl --socks5-hostname 127.0.0.1:1080 --proxy-user alice:secret
http://127.0.0.1:8000/64k.bin -o /dev/null // conexion via proxy

time curl http://127.0.0.1:8000/64k.bin -o /dev/null // conexion directa
```

Código 4. Comandos para enviar el pedido por un archivo de 64KB via proxy y de manera directa (respectivamente).

Los resultados de esta prueba están demostrados en la figura XX. Lo que se puede concluir de estos resultados es que el servidor proxy se mantiene bastante uniforme para tamaños de buffer de un rango menor, pero para tamaños más grandes, la diferencia entre la conexión directa y la conexión vía proxy se pronuncia mucho más y los tiempos ya no se mantienen tan uniformes.

8. Guía de instalación

8.1 Requisitos

Para poder instalar y compilar el servidor, se requieren dos herramientas principales: `make` y `gcc`.

8.2 Compilacion

Al utilizar un Makefile, la compilación se logra corriendo los siguientes dos comandos: `make clean` (opcional, pero recomendado para asegurarnos de que se compile la última versión) y `make all`.

8.3 Ejecucion

El ejecutable se genera en la carpeta bin. Por lo tanto, si estamos parados en la raíz del directorio, para correrlo debemos correr:

```
> ./bin/server [OPTIONS]
```

Código 5. Comando para correr el servidor.

Las opciones son detalladas en la siguiente sección.

9. Instrucciones para la configuración

9.1 Servidor

Por el lado del servidor, se pueden setear diversos parámetros al momento de ejecutarlo. Estos pueden ser:

- `-h` Imprime la ayuda y termina
- `-l <SOCKS addr>` Dirección donde se servirá el servidor
- `-L <MNG addr>` Dirección donde se servirá el manager
- `-p <SOCKS port>` Puerto donde se servirá el servidor
- `-P <MNG port>` Puerto donde se servirá el manager
- `-u <usr>:<pass>` Usuario y contraseña que se puede usar
- `-v` Imprime información sobre la versión

9.2 Cliente

La herramienta de monitoreo permite realizar cambios en tiempo de ejecución (como agregar o eliminar usuarios) y consultar métricas sin detener el servidor. Para su funcionamiento, se debe indicar la dirección del servidor de gestión y las credenciales de un administrador. Para ejecutarlo se debe correr:

```
> ./bin/client <host:port> -l <admin:pass> [ACCIÓN]
```

Código 6. Comando para correr la aplicación de cliente.

Los parámetros y las acciones son los siguientes:

- `<host:port>`: Dirección y puerto donde está corriendo el servicio de gestión (debe coincidir con los parámetros `-L` y `-P` del servidor). Debe ser necesariamente el primer parámetro.
- `-l <user:pass>`: Credenciales de login. Se requieren permisos de administrador para ejecutar comandos. Este parámetro es obligatorio.
- `-a <user:pass>`: Da de alta un nuevo usuario con privilegios estándar (navegación).
- `-A <user:pass>`: Da de alta un nuevo usuario con privilegios de administrador (acceso a esta herramienta).
- `-d <user>`: Elimina un usuario existente del registro.
- `-c <user:newpass>`: Modifica la contraseña de un usuario existente.
- `-m <method_1> ... <method_n>`: Modifica los métodos de autenticación aceptados en tiempo de ejecución (0 para NO AUTH, 2 para USER/PASS).
- `-s`: Solicita y muestra las estadísticas de uso del servidor (bytes transferidos, conexiones, etc.).
- `-h`: Muestra el mensaje de ayuda.

10. Ejemplos de configuración y monitoreo

A continuación, se presentan una serie de casos de uso prácticos utilizando la herramienta de administración. Para estos ejemplos, se asume que el servicio de gestión del servidor está escuchando en la dirección `127.0.0.1` puerto `8080`, y que existe un usuario administrador predeterminado con credenciales `admin:admin`.

10.1 Gestión de usuarios

Esta herramienta permite el agregado de nuevos clientes mientras el servidor se está ejecutando, para permitirles navegar a través del proxy. El siguiente comando muestra cómo agregar un nuevo usuario llamado "juan" con contraseña "1234", otorgándole permisos de navegación estándar:

```
> ./bin/client 127.0.0.1:8080 -l admin:admin -a juan:1234
```

Código 7. Ejemplo de agregado de un nuevo usuario con la aplicación de cliente.

Cuya respuesta se puede ver de la siguiente manera:

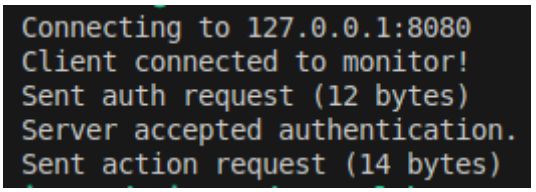

Figura 5. Ejemplo de respuesta al agregar un nuevo usuario con la aplicación de cliente.

Si posteriormente se desea actualizar la seguridad de este usuario, es posible modificar su contraseña. Por ejemplo, para cambiar la clave de "juan" a "nuevaClave":

```
./bin/client 127.0.0.1:8080 -l admin:admin -c juan:nuevaClave
```

Código 8. Ejemplo de modificación de contraseña con la aplicación de cliente.

Al correr este comando correctamente, deberíamos obtener una respuesta equivalente a la anterior, con la cantidad de bytes correspondiente.

Finalmente, si el usuario ya no debe tener acceso al sistema, se puede eliminar su registro de la base de datos en memoria mediante el comando de baja, también obteniendo una respuesta equivalente a la anterior:

```
./bin/client 127.0.0.1:8080 -l admin:admin -d juan
```

Código 9. Ejemplo de eliminación de un usuario con la aplicación de cliente.

10.2 Monitoreo del servidor

Para evaluar el rendimiento del proxy en tiempo real, el administrador puede solicitar las métricas de uso. Al ejecutar la consulta de estadísticas, se recuperan los contadores internos del servidor y los formatea para su visualización. El comando para realizar esta solicitud es el siguiente:

```
./bin/client 127.0.0.1:8080 -l admin:admin -s
```

Código 10. Comando para consultar las métricas actuales del servidor desde la aplicación cliente.

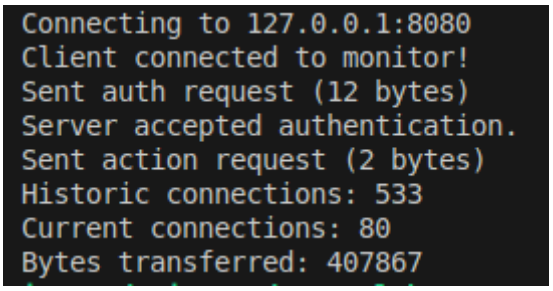

Figura 6. Ejemplo de respuesta al solicitar las métricas del servidor con la aplicación cliente.

Estos valores reflejan, respectivamente, la cantidad total de clientes que han pasado por el servidor desde su inicio, cuántos están conectados en este preciso instante y el volumen total de tráfico procesado.

10.3 Gestión de métodos de autenticación

Además de la gestión de usuarios, es posible modificar la seguridad del servidor en tiempo real alterando los métodos de autenticación permitidos según el RFC 1928.

Para permitir acceso libre (sin autenticación):

```
./bin/client 127.0.0.1:8081 -l admin:admin123 -m 0
```

Código 11. Comando para cambiar el método de autenticación a No Auth desde la aplicación cliente.

Para volver a un estado estricto (solo usuario y contraseña):

```
./bin/client 127.0.0.1:8081 -l admin:admin123 -m 2
```

Código 12. Comando para cambiar el método de autenticación a User/Password desde la aplicación cliente.

Para permitir ambos métodos simultáneamente:

```
./bin/client 127.0.0.1:8081 -l admin:admin123 -m 0 2
```

Código 13. Comando para cambiar que el método de autenticación pueda ser NoAuth y User/Password desde la aplicación cliente.

11. Arquitectura de la aplicación

Como ya fue descrito anteriormente, la arquitectura del servidor se basa en un modelo de manejo de eventos que se ejecutan en un único *thread*, multiplexando la entrada y salida (a través del módulo `selector`). Esto permite manejar múltiples conexiones de manera concurrente y eficiente. El sistema mencionado se estructura en tres componentes principales:

1. El *selector*: Es el encargado de monitorear todos los sockets activos (clientes y servidores remotos). Su función es notificar al programa principal únicamente cuando un socket está listo para leer o escribir, evitando esperas innecesarias.
2. Los *handlers*: Son las funciones que reaccionan ante los eventos del selector. Se encargan de recibir los datos y decidir qué hacer con ellos.
3. La máquina de estados: Dado que la ejecución no puede detenerse, la lógica del protocolo SOCKS v5 se dividió en pequeños pasos o "estados" (2.2.1 Servidor proxy). La máquina de estados mantiene el registro de en qué paso se encuentra cada cliente, permitiendo procesar la información de a fragmentos sin perder el contexto.

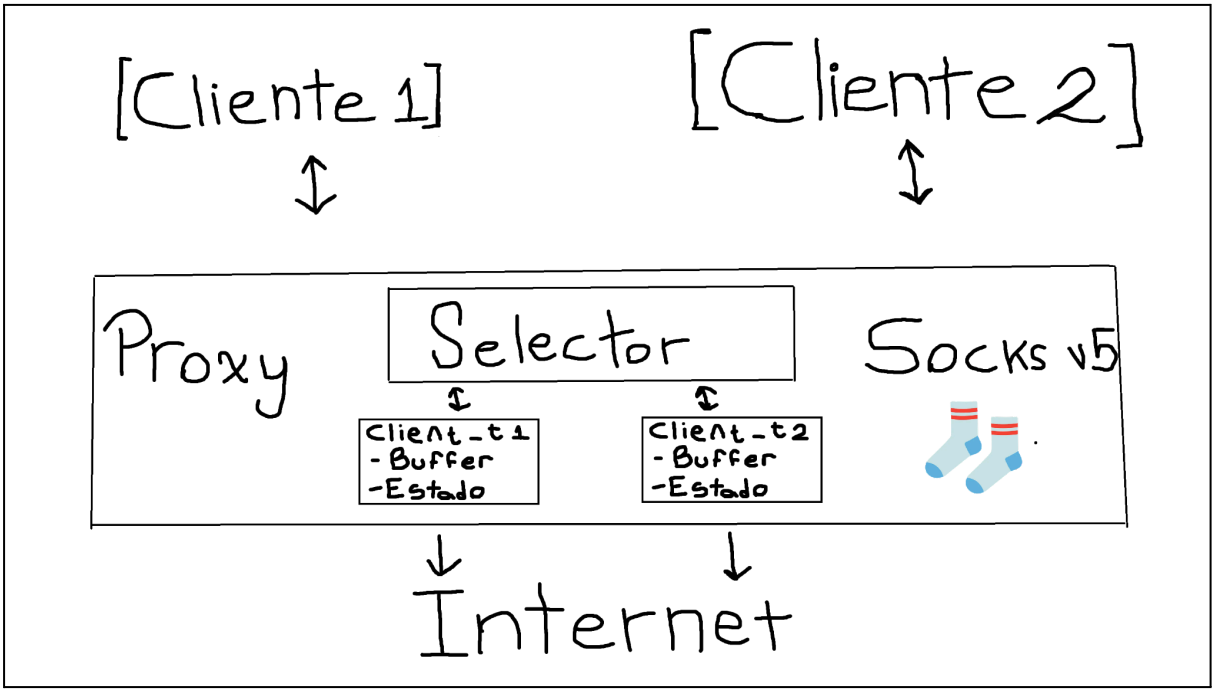

Figura 7. Diagrama de diseño simplificado de la aplicación servidor.

12. Referencias

[1] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones. *SOCKS Protocol Version 5*. RFC 1928, IETF, Marzo 1996. https://www.rfc-editor.org/rfc/rfc1928.txt

[2] M. Leech. *Username/Password Authentication for SOCKS V5*. RFC 1929, IETF, Marzo 1996. Disponible en: https://www.rfc-editor.org/rfc/rfc1929.txt

[3] The Linux Man-Pages Project. *getaddrinfo_a(3) — Asynchronous name resolution*. Linux Programmer's Manual. https://man7.org/linux/man-pages/man3/getaddrinfo_a.3.html