Trabajo Práctico Especial

Protocolos de Comunicación

Grupo 2

Primer cuatrimestre de 2025

Integrantes:

Bassi, Santiago	Legajo: 64643
Hillar, Conrado	Legajo: 64633
Maruottolo, Ignacio	Legajo: 64611
Testoni, Ezequiel	Legajo: 64709

Fecha de entrega: Martes 15 de Julio de 2025

ÍNDICE

1. Introducción **2**

2. Protocolos y aplicaciones desarrolladas **2**

2.1. Servidor proxy SOCKS5 2

2.2. Protocolo de administración S5ADMIN 2

3. Problemas encontrados durante el diseño y la implementación **3**

4. Limitaciones de la aplicación **4**

5. Posibles extensiones **4**

6. Conclusiones **5**

7. Ejemplos de prueba **5**

7.1. Velocidad en función del buffer_size 5

7.2. Stress test de usuarios 6

7.3. Navegación utilizando el proxy 8

8. Guía de instalación **8**

9. Instrucciones para la configuración **8**

9.1. Configuración del Servidor 8

9.2. Configuración del Cliente de Administración 9

10. Ejemplos de configuración y monitoreo **9**

10.1. Comandos Generales 9

10.2. Comandos de Monitoreo 10

10.3. Comandos de Configuración 10

11. Documento de diseño del proyecto **11**

11.1. Componentes Principales 11

11.2. Flujo de una Conexión SOCKSv5 11

1. Introducción

El trabajo práctico especial de la materia Protocolos de Comunicación consistió en la implementación de un servidor SOCKS5 respetando el RFC 1928, el desarrollo de un protocolo de monitoreo y configuración de este servidor (llamado S5ADMIN) y la puesta en funcionamiento de un cliente que utilice este protocolo. En el presente informe se detalla el proceso de creación del servidor y el protocolo recién mencionados.

2. Protocolos y aplicaciones desarrolladas

2.1. Servidor proxy SOCKS5

Se utilizó el protocolo SOCKS5 (definido en la RFC 1928) como mecanismo de encapsulamiento entre aplicaciones cliente y servidores a través de firewalls o redes con políticas restrictivas. Este protocolo actúa como una capa de abstracción entre la capa de aplicación y la de transporte, permitiendo que las aplicaciones puedan establecer conexiones TCP o enviar datagramas UDP de forma transparente, sin necesidad de conocer la topología real de red. Además, SOCKS5 incorpora soporte para autenticación y direccionamiento flexible (incluyendo IPv4, IPv6 y nombres de dominio), lo que lo convierte en una solución adecuada para entornos donde se requiere control de acceso, compatibilidad con múltiples protocolos y seguridad en la administración del tráfico saliente.

Por diseño, la implementación desarrollada se limita al comando CONNECT de SOCKS5; es decir, actúa exclusivamente como proxy de flujo TCP. El comando UDP ASSOCIATE, que habilita el reenvío de datagramas UDP en la especificación original, no se implementó porque la consigna especifica: “Alcance del proyecto: el objetivo principal es facilitar conexiones de salida basadas en TCP.” En consecuencia, cualquier cliente que intente realizar un UDP ASSOCIATE recibirá la respuesta “Comando no soportado” (REP = 0x07), tal como lo establece la RFC 1928.

El servidor utiliza un modelo de I/O no bloqueante multiplexada mediante la API de selector provista por la cátedra, lo que le permite manejar un gran número de conexiones concurrentes de manera eficiente. La arquitectura se divide en tres componentes principales:

- **Core del Servidor (server.c)**: Responsable de inicializar el selector, abrir los sockets de escucha (para SOCKSv5 y para S5ADMIN), y entrar en el ciclo principal de manejo de eventos. Gestiona las señales del sistema para garantizar la correcta liberación de recursos al apagarse.

- **Módulo SOCKSv5 (socks5.c)**: Implementa el protocolo SOCKSv5 (RFC 1928) y la autenticación por usuario/contraseña (RFC 1929). Utiliza una máquina de estados para gestionar el ciclo de vida de cada conexión de cliente, desde el handshake inicial hasta el intercambio de datos.

- **Módulo de Administración (s5admin.c)**: Implementa el protocolo de administración S5ADMIN (RFC adjunto en el proyecto) basado en texto sobre un socket TCP separado. Permite la configuración y monitoreo del servidor en tiempo de ejecución.

2.2. Protocolo de administración S5ADMIN

Se implementó un protocolo de aplicación para la configuración y administración del servidor proxy en tiempo real. Se decidió que dicho protocolo funcione sobre TCP, ya que al ser orientado a

conexión y confiable, simplifica la implementación abstrayendo el orden en el que llegan los paquetes y la lógica de reenvío de los mismos en caso de pérdida.
El protocolo permite a un administrador conectarse al servidor y realizar diversas tareas de configuración y monitoreo. Las posibles tareas que se permiten realizar son:

- Añadir un nuevo usuario.
- Eliminar un usuario.
- Listar todos los usuarios.
- Mostrar las métricas del servidor (bytes transferidos, conexiones históricas y conexiones actuales).
- Mostrar un registro de las conexiones manejadas por el servidor.
- Establecer el nivel de logging del servidor.
- Establecer el número máximo de conexiones simultáneas (limitado a 512 como máximo) .
- Establecer el tamaño del buffer de lectura/escritura.
- Mostrar la configuración actual del servidor.
- Listar los comandos disponibles.
- Comprobar si el servidor está activo.
- Cerrar la conexión de administración.

3. Problemas encontrados durante el diseño y la implementación

Durante el desarrollo del proyecto surgieron diversos problemas, que fueron desde cuestiones de organización y modularización del código hasta inconvenientes relacionados con el rendimiento del servidor.

Uno de los primeros grandes desafíos fue que las pocas líneas iniciales de código crecieron rápidamente hasta convertirse en miles, sin una estructura de archivos y carpetas adecuada. Esto derivó en un código difícil de leer, mantener y extender. Para solucionarlo, se realizó una refactorización completa del proyecto, en la que se aprovechó para modularizar funciones y reorganizar la estructura. Por ejemplo, se crearon archivos como socks5_responses.c, que encapsulan la lógica de generación de respuestas del protocolo SOCKS5, permitiendo obtener cada respuesta con una simple llamada a función.

Otra dificultad significativa apareció al momento de realizar las primeras pruebas del servidor. El sistema presentaba numerosos errores y quedaba colgado, y esos errores eran difíciles de identificar. Para resolver este problema, se implementó un sistema de logging con distintos niveles (debug, info, warning, error), lo cual facilitó exponencialmente la localización y el análisis de fallos durante la ejecución.

También se detectó un problema importante relacionado con el bajo rendimiento del servidor. A través de pruebas específicas, se observó que el proxy generaba una considerable ralentización en la transferencia de datos. Para mejorar esta situación, se tomaron varias medidas, entre ellas:

Se modificó el flujo del selector. Originalmente, el flujo era select → read → select → write, lo cual resultaba ineficiente. Fue reemplazado por select → read → write → select, logrando reducir la latencia y mejorar el throughput general. Luego, se verificó usando **trace** que el cambio generaba una mejora significativa.

Se ajustó el tamaño por defecto de los buffers, con el objetivo de reducir la frecuencia con la que el servidor debía volver al select, minimizando así el overhead del sistema operativo.

4. Limitaciones de la aplicación

A pesar de cumplir con los requisitos fundamentales, la implementación actual presenta ciertas limitaciones que son importantes de destacar:

Soporte parcial del protocolo SOCKSv5: La implementación actual solo soporta el comando CONNECT para túneles TCP. Los comandos BIND y UDP ASSOCIATE, especificados en el RFC 1928, no han sido implementados. Esto significa que el proxy no puede ser utilizado para aplicaciones que requieran establecer conexiones entrantes (como FTP en modo activo) o para tráfico UDP.

Escalabilidad limitada por pselect(): El servidor utiliza la llamada al sistema pselect() para la multiplexación de I/O. Aunque es portable, pselect() está limitada por el valor de FD_SETSIZE (1024 fds), lo que impone un límite superior teórico en la cantidad de conexiones concurrentes que el servidor puede manejar.

Configuración y usuarios volátiles: Toda la configuración del servidor, incluyendo la lista de usuarios y sus contraseñas, se almacena únicamente en memoria. Esto implica que si el servidor se reinicia, todos los cambios realizados en tiempo de ejecución a través del panel de administración se pierden. No hay un mecanismo de persistencia para guardar el estado en un archivo de configuración.

Protocolo de administración no seguro: El protocolo de administración (S5Admin) se transmite en texto plano. Esto significa que las credenciales de los usuarios y toda la información de configuración y monitoreo viajan sin cifrar por la red. Esto representa un riesgo de seguridad significativo si el cliente de administración se utiliza desde una red no confiable, ya que un atacante podría capturar el tráfico y obtener control total sobre el proxy.

Sin autenticación en el panel de administración: El acceso al panel de administración no está protegido por ninguna forma de autenticación. Cualquier persona que pueda conectarse al puerto de administración puede ejecutar comandos.

Únicamente se puede utilizar el método de autenticación de user/pass de SOCKS5, no se cuenta con soporte para GSSAPI por ejemplo.

5. Posibles extensiones

Para superar las ~500 conexiones concurrentes, sería necesario migrar a mecanismos más modernos y eficientes como epoll en Linux o kqueue en sistemas BSD/macOS, que no tienen esta limitación y ofrecen un rendimiento O(1) en lugar de O(n).

Además, agregar un sistema de almacenamiento persistente de los registros de usuarios permitiría una utilización más dinámica y “real” del servidor, pues no se tendrían que agregar los usuarios nuevamente cada vez que se reinicia el servidor.

6. Conclusiones

El desarrollo de este proyecto ha permitido alcanzar con éxito los objetivos fundamentales planteados en la consigna. Se ha logrado implementar un servidor proxy SOCKSv5 funcional, concurrente y robusto, junto con un panel de administración que permite la configuración y el monitoreo en tiempo de ejecución, demostrando una sólida comprensión de la programación de redes en C, el manejo de I/O no bloqueante y el diseño de protocolos de aplicación.

La elección de una arquitectura basada en una máquina de estados para el protocolo SOCKSv5 y el uso de buffers circulares para la gestión de la I/O fueron decisiones de diseño cruciales que permitieron manejar la complejidad inherente al proyecto, resultando en un código más limpio, modular y mantenible. Asimismo, la implementación del protocolo de administración S5Admin, aunque es simple, cumple eficazmente su propósito de facilitar la configuración y monitoreo del servidor de manera dinámica.

Si bien la implementación actual tiene limitaciones conocidas, como el soporte parcial del estándar SOCKSv5 y la falta de persistencia y seguridad avanzada, el proyecto constituye una base sólida y bien estructurada. Las posibles extensiones delineadas abren un camino claro para futuras mejoras que podrían convertir esta implementación en una solución de nivel de producción.

En definitiva, el trabajo práctico ha sido una valiosa experiencia de aprendizaje que ha permitido aplicar conceptos teóricos de protocolos de comunicación en un escenario práctico y desafiante.

7. Ejemplos de prueba

7.1. Velocidad en función del buffer_size

Para determinar el tamaño del buffer, se creó un script de bash que itera por varios tamaños de buffer distintos (entre 512 B y 512 KB). Además, el script genera varios archivos de tamaño entre 1 MB y 128 MB, y los sirve en un servidor HTTP local (de esta manera eliminando la inconsistencia y fluctuaciones en la velocidad de la red). Luego, itera por todas las combinaciones posibles de tamaño de buffer y tamaño de archivo, testeando con el servidor proxy y sin él, repitiendo varias veces (10 por default) cada combinación y tomando un promedio. La información pertinente se almacena en un archivo .csv para su posterior análisis.

# buffer_size_B	file_size_MB	# direct_avg_time_s	# direct_avg_speed_MBps	# proxy_avg_time_s	# proxy_avg_speed_MBps	%_proxy_vs_direct
Count: 8	Avg: 31.88	Avg: 0.01	Avg: 2290.84	Avg: 0.03	Avg: 793.48	Avg: 37.00%
4096	1	0.0013	870.8361	0.0024	436.6676	50.14%
4096	2	0.0017	1360.0173	0.0033	624.9287	45.95%
4096	4	0.0025	1733.2467	0.0077	607.8707	35.07%
4096	8	0.0036	2316.989	0.0103	789.4731	34.07%
4096	16	0.0061	2666.9355	0.0183	884.5196	33.17%
4096	32	0.0108	2987.0653	0.0334	959.3695	32.12%
4096	64	0.0203	3168.8371	0.064	1001.5777	31.61%
4096	128	0.0399	3222.8194	0.1231	1043.4582	32.38%

Figura 1 - Mediciones realizadas con un buffer de 4 KB

La relación entre el tamaño del buffer y la velocidad de descarga parece tener un comportamiento logarítmico. Esto se puede observar por ejemplo en el gráfico de velocidad en función del tamaño del buffer para un archivo de 128MB:

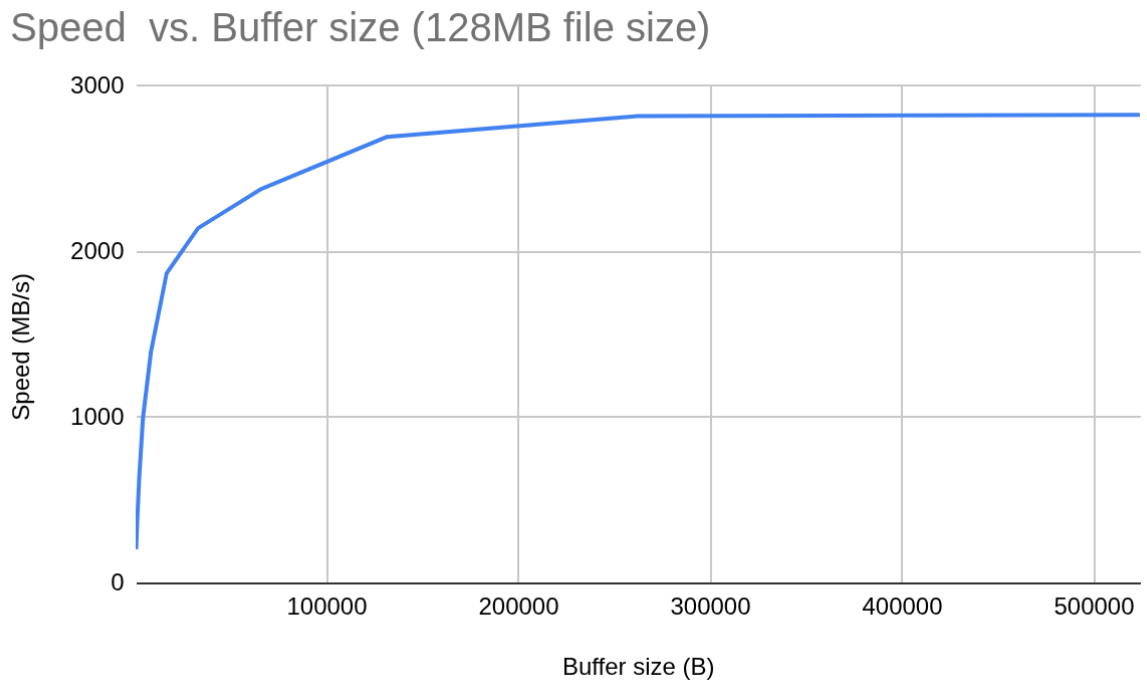

Figura 2 - Mediciones de velocidad de transferencia (en MB/s) en función del tamaño del buffer (en bytes), para un archivo de 128 MB.

Se puede observar que luego de los 128 KB de tamaño de buffer, no hay mejoras significativas.

Para un análisis más realista, se testearon varios tamaños de archivo distintos y se tomó el promedio de todas las velocidades para cada tamaño de buffer. De esta forma, se observó que el tamaño ideal era bastante consistente entre los distintos tamaños de archivo, por lo que se decidió establecer el tamaño default en 128KB.

7.2. Stress test de usuarios

Se creó un script de bash que permite testear varias conexiones al proxy de manera simultánea, haciendo uso de las herramientas “proxychains” y “wrk”. El script permite definir la cantidad de usuarios a testear, la cantidad de hilos de ejecución en las que se realiza el test y la URL a la cual se quiere conectar. De esta manera, se testeó que puedan conectarse de manera simultánea 500 usuarios. La herramienta “wrk” además muestra información útil de rendimiento. Haciendo uso de ésta, se hizo un análisis de la degradación de la performance en relación a la cantidad de usuarios. La prueba se hizo de manera controlada, nuevamente levantando un servidor HTTP local que sirve un archivo de 2MB. Cada uno de los usuarios hizo curl a dicho archivo.

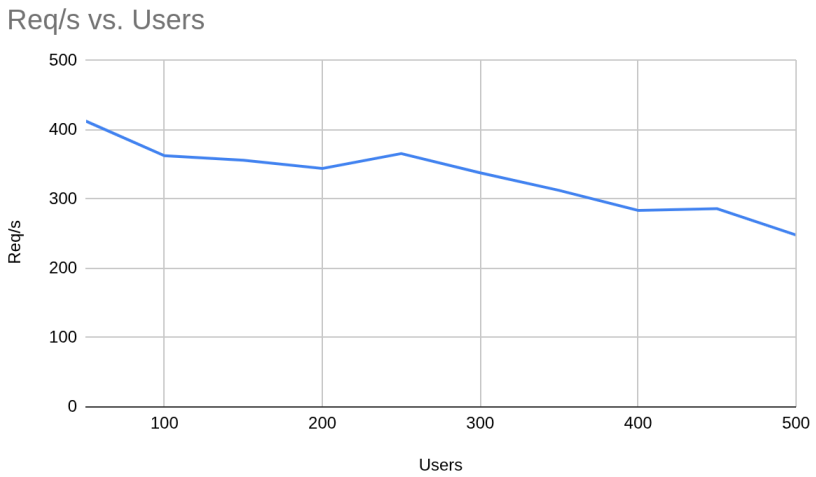

Figura 3 - Mediciones de requests/s en función de la cantidad de conexiones concurrentes.

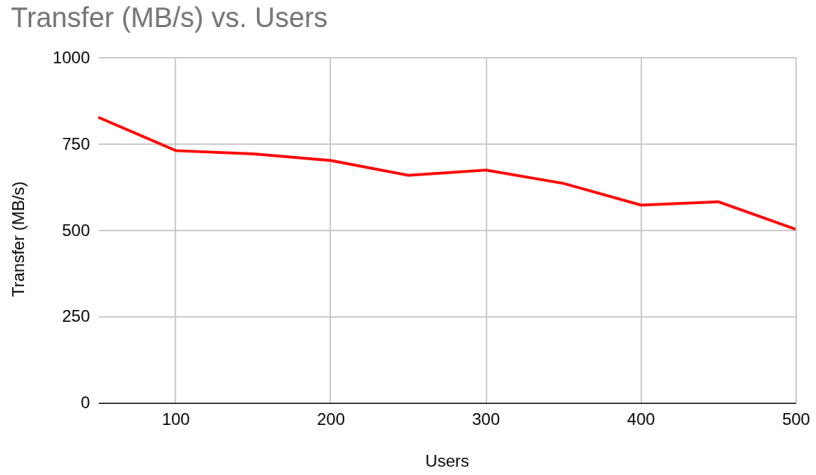

Figura 4 - Mediciones de velocidad de transferencia (en MB/s) en función de la cantidad de conexiones concurrentes.

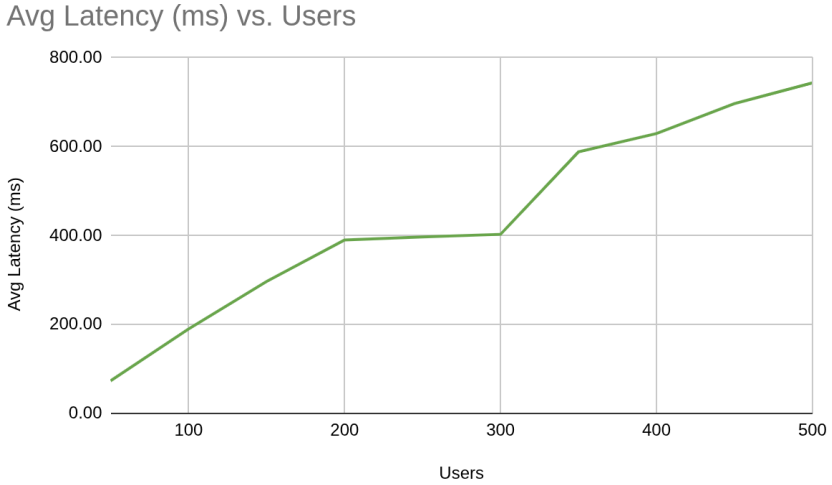

Figura 5 - Mediciones de latencia (en ms) en función de la cantidad de conexiones concurrentes.

Se observa que a medida que aumenta la cantidad de usuarios concurrentes, la cantidad de requests por segundo y la velocidad de transferencia disminuyen, como también aumenta la latencia. Todos estos son indicios de una degradación en el rendimiento del servidor, lo cual es esperable considerando que cada usuario aumenta la cantidad de recursos que el servidor utiliza.

7.3. Navegación utilizando el proxy

Otra prueba realizada fue navegar con Google Chrome configurando nuestro servidor como el proxy. Esto se puede realizar con el comando:

google-chrome --proxy-server="socks5://127.0.0.1:1080"
--user-data-dir=/tmp/chrome-proxy-test --no-first-run --no-default-browser-check

Esto abre una nueva ventana de Chrome que redirige todo el tráfico a través del proxy. Se probó ver videos y abrir distintas páginas, y todo funcionaba con total normalidad, sin errores de ningún tipo.

Además se testeó, utilizando herramientas como curl, realizar peticiones HTTP a distintos sitios (protos.foo, google.com.ar, api.ipify.org, etc.).

8. Guía de instalación

El código fuente incluye un archivo Makefile que facilita la compilación del proyecto. Una vez descomprimido el directorio, desde la carpeta raíz se puede compilar utilizando el siguiente comando:

$ make <target> [MODE=release]

donde <target> puede ser: *all*, *server* o *client*, dependiendo de los binarios que se desee generar. El parámetro MODE=release es opcional: si se lo omite, la compilación se realiza con herramientas de análisis como fsanitize=address, útiles para debugging; si se lo incluye, se compila y linkea en modo producción, sin esas herramientas.

Es necesario contar con el compilador GCC instalado para poder ejecutar este comando. Los binarios generados (socks5d y/o admin_client) se ubicarán en el directorio bin, en la raíz del proyecto.

9. Instrucciones para la configuración

La configuración inicial del servidor y la conexión del cliente de administración se realizan mediante argumentos de línea de comandos. No se utilizan archivos de configuración.

9.1. Configuración del Servidor

El servidor se configura al momento de su ejecución. A continuación se detallan las opciones disponibles:

-h: Muestra un mensaje de ayuda con todas las opciones disponibles y termina la ejecución.
-l <addr>: Especifica la dirección en la que el servidor SOCKSv5 escuchará las conexiones de los clientes. Si no se especifica, por defecto es 0.0.0.0 (escucha en todas las interfaces).
-L <addr>: Especifica la dirección en la que el servidor de administración escuchará las conexiones. Si no se especifica, por defecto es 127.0.0.1 (solo escucha en localhost).

-p <port>: Especifica el puerto para el servicio SOCKSv5. El valor por defecto es 1080.
-P <port>: Especifica el puerto para el servicio de administración. El valor por defecto es 8080.
-u <user:pass>: Define un usuario y su contraseña para la autenticación en el proxy. Esta opción se puede especificar múltiples veces (hasta un máximo de 10) para crear varios usuarios. Importante: No debe haber espacios antes o después del carácter ':'.
-v: Muestra la versión del servidor y termina la ejecución.
-g <DEBUG | INFO | WARNING | ERROR>: Especifica el nivel de logueo. Por default el nivel es INFO.
-f <archivo>: Especifica el archivo al cual se quieren escribir los logs. Si se omite, los logs se imprimen por salida estándar.

Ejemplo de ejecución:

Ejecutar el servidor en el puerto 1080 para SOCKS5 y en el 8080 para administración. Se crean dos usuarios: 'user1' con contraseña 'pass1' y 'user2' con contraseña 'pass2'.

./bin/socks5d -p 1080 -P 8080 -u user1:pass1 -u user2:pass2

9.2. Configuración del Cliente de Administración

El cliente de administración necesita saber cómo conectarse al servidor.

-**h:** Muestra un mensaje de ayuda con todas las opciones disponibles y termina la ejecución.
-L <addr>: Especifica la dirección del servidor de administración al que se conectará. El valor por defecto es 127.0.0.1.
-**P <port>:** Especifica el puerto del servidor de administración. El valor por defecto es 8080.
-v: Muestra la versión del cliente.
-g <ERROR | WARNING | INFO | DEBUG>: Especifica el nivel de logueo. Por default el nivel es INFO.
-f <archivo>: Especifica el archivo al cual se quieren escribir los logs. Si se omite, los logs se imprimen por salida estándar.

Ejemplo de ejecución:

Conectar el cliente al servidor de administración que corre en localhost en el puerto 8080.

./bin/admin_client -L 127.0.0.1 -P 8080

10. Ejemplos de configuración y monitoreo

El panel de administración es la herramienta para interactuar con el servidor en tiempo real. A continuación, se detallan todos los comandos disponibles.

10.1. Comandos Generales

- **`HELP`**: Muestra la lista de todos los comandos disponibles.
- **`PING`**: Envía una señal al servidor para verificar que la conexión está activa. El servidor responderá PONG.
- **`EXIT`**: Cierra la conexión con el servidor de administración.

10.2. Comandos de Monitoreo

- **`GET_METRICS`**: Devuelve las métricas de operación actuales.

  ```
  >: GET_METRICS
  CONNECTED: 5
  HISTORICAL: 152
  BYTES: 81920
  ERRORS: 0
  END
  ```

- **`GET_ACCESS_REGISTER`**: Muestra un registro de los accesos de los usuarios.

```
>: GET_ACCESS_REGISTER
Fecha                   Usuario  Tipo   IP_origen       Pto_org Destino         Pto_dst Status
2025-07-13T18:20:00Z    user1    A      192.168.0.2     54786   www.example.com 443     0
2025-07-13T18:21:15Z    user2    A      ::1             12345   www.google.com  443     0
END
```

- **`GET_CONFIG`**: Muestra la configuración actual del servidor.

  ```
  >: GET_CONFIG
  LOG_LEVEL: INFO
  BUFFER_SIZE: 1024
  MAX_CONN: 500
  END
  ```

10.3. Comandos de Configuración

- **`LIST_USERS`**: Muestra la lista de todos los usuarios configurados.

- **`ADD_USER <usuario> <contraseña>`**: Agrega un nuevo usuario.

  ```
  >: ADD_USER user_temp 12345
  OK
  ```

- **`REMOVE_USER <usuario>`**: Elimina un usuario existente.

  ```
  >: REMOVE_USER user_temp
  OK
  ```

- **`SET_LOGLEVEL <nivel>`**: Cambia el nivel de verbosidad de los logs. Niveles: DEBUG, INFO, WARNING, ERROR.

  ```
  >: SET_LOGLEVEL DEBUG
  OK
  ```

- **`SET_BUFF <bytes>`**: Cambia el tamaño de los buffers de I/O para las nuevas conexiones.

  ```
  >: SET_BUFF 8192
  OK
  ```

- **`SET_MAX_CONN <cantidad>`**: Cambia el número máximo de conexiones concurrentes permitidas.

```
>: SET_MAX_CONN 1000
OK
```

11. Documento de diseño del proyecto

La arquitectura del servidor está diseñada en torno a un ciclo de eventos centralizado y sin bloqueo, manejado por un selector de I/O. Este diseño permite un alto grado de concurrencia con un bajo consumo de recursos.

11.1. Componentes Principales

Selector (selector.c): Es el núcleo de la concurrencia. Utiliza select() para monitorear múltiples descriptores de archivo (sockets) y detectar cuándo están listos para operaciones de lectura o escritura sin bloquear el proceso. Mantiene un registro de todos los sockets activos y los handlers de eventos (fd_handler) asociados a cada uno.

Handlers de Eventos (fd_handler): Son conjuntos de punteros a funciones (handle_read, handle_write, handle_close) que definen cómo reaccionar a los eventos de un descriptor de archivo. Cada socket registrado en el selector tiene un fd_handler asociado.

Máquina de Estados SOCKSv5 (socks5_stm.c): Abstrae la lógica del protocolo SOCKSv5. Cada conexión de cliente tiene su propia instancia de esta máquina de estados, que avanza a través de las fases del protocolo (handshake, autenticación, solicitud, copia de datos) en respuesta a los datos recibidos del cliente.

Módulo de Administración (s5admin.c): Implementa la lógica para el protocolo de configuración. Funciona de manera similar a una conexión SOCKSv5, con sus propios handlers de eventos para leer comandos y escribir respuestas.

Buffers (buffer.c): Proporcionan una capa de abstracción sobre la I/O de red, manejando de forma transparente las lecturas y escrituras parciales para liberar de esta responsabilidad al resto de la aplicación.

11.2. Flujo de una Conexión SOCKSv5

1. Aceptación: El selector notifica al socket de escucha principal que hay una nueva conexión entrante. La función socksv5_passive_accept acepta la conexión, crea un nuevo socket para el cliente y lo registra en el selector con los handlers de eventos de SOCKSv5.

2. Máquina de Estados: A partir de aquí, la socks5_stm toma el control del flujo de la conexión.

- Estado de Handshake: Lee el saludo del cliente, selecciona un método de autenticación y escribe la respuesta.
- Estado de Autenticación: Lee las credenciales del cliente, las valida y escribe la respuesta.
- Estado de Solicitud: Lee la solicitud de conexión del cliente (ej. CONNECT a example.com:80), resuelve el destino y establece una conexión no bloqueante con el servidor de origen.

- Estado de Copia (Copy): Una vez que todo está conectado, la máquina de estados entra en su fase final. En este punto, simplemente transfiere datos en ambas direcciones: del cliente al origen y del origen al cliente, utilizando los buffers para mediar el flujo.

3. Cierre: Si cualquier parte de la comunicación se cierra (cliente o servidor de origen), los handlers de cierre correspondientes se encargan de liberar todos los recursos asociados a la conexión (sockets, buffers, máquina de estados) y de dar de baja los file descriptors del selector.

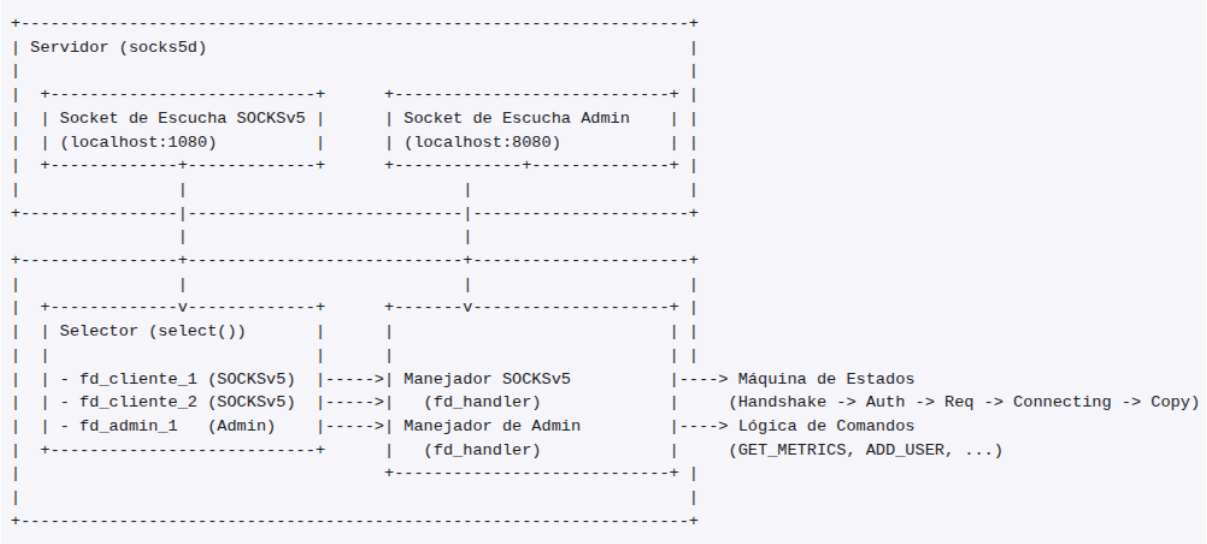

Figura 6 - Diagrama del flujo del servidor proxy SOCKSv5