Trabajo práctico especial

Servidor proxy para el protocolo SOCKSv5[RFC1928]

Protocolos de la comunicación 72.07

Grupo 14

Integrante	Legajo
lstupnik@itba.edu.ar	64233
mrossiseifert@itba.edu.ar	63202
nbellavitis@itba.edu.ar	64001
vsanguinetti@itba.edu.ar	63098

Tabla de contenidos:

1. DESCRIPCIÓN DETALLADA DE LOS PROTOCOLOS Y APLICACIONES DESARROLLADAS

1.1 Arquitectura General del Sistema

El proyecto implementa un servidor proxy SOCKS5 completo con capacidades de administración remota, desarrollado en C siguiendo las especificaciones RFC 1928. La arquitectura se basa en un patrón de máquina de estados que maneja múltiples conexiones concurrentes de forma no bloqueante utilizando un selector I/O. El sistema está compuesto por tres aplicaciones principales: el servidor SOCKS5, un cliente de pruebas SOCKS5 y un cliente de administración, todos desarrollados con el objetivo de proporcionar una solución completa y funcional para el tunneling de conexiones TCP a través de un proxy.

1.2 Protocolo SOCKS5 (RFC 1928)

El protocolo SOCKS5 es el núcleo del sistema y permite a los clientes establecer conexiones TCP a través del proxy hacia destinos remotos. El protocolo funciona en fases secuenciales: primero se realiza una negociación donde el cliente propone métodos de autenticación disponibles (sin autenticación o con usuario/contraseña), luego se ejecuta la autenticación si es necesaria, y finalmente se procesa la solicitud de conexión especificando el destino deseado. El servidor responde con el resultado de la operación y, en caso de éxito, comienza a transferir datos bidireccionalmente entre el cliente y el destino.

El protocolo soporta múltiples tipos de dirección de destino: direcciones IPv4, IPv6 y nombres de dominio, que son resueltos automáticamente por el servidor. Actualmente solo se implementa el comando CONNECT para conexiones TCP, que es el más comúnmente utilizado. El sistema maneja la resolución DNS de forma asíncrona para evitar bloquear otras conexiones durante la búsqueda de nombres de dominio. En caso de que la primera resolución falle se intentara con las siguientes que retorne la función getaddrinfo_a.

1.3 Protocolo de Management

El protocolo de management es una extensión personalizada que permite administrar el servidor SOCKS5 de forma remota. Este protocolo funciona sobre TCP y requiere autenticación previa como administrador antes de permitir cualquier operación. Una vez autenticado, el administrador puede consultar estadísticas del servidor (conexiones activas, bytes transferidos, etc.), gestionar usuarios autorizados (agregar, eliminar, cambiar contraseñas), y modificar configuraciones dinámicas como el tamaño de los buffers o el método de autenticación global.
El protocolo utiliza un formato simple de mensajes con versión, comando, longitud de payload y datos opcionales. Los comandos incluyen operaciones de autenticación, consulta de estadísticas, gestión de usuarios y configuración del sistema. Las respuestas incluyen códigos de estado que indican el éxito o fracaso de la operación, junto con información adicional cuando es relevante.

1.3.1 Especificación del Protocolo de Management

Notas preliminares de notación:

- Cuando un octeto debe tomar un valor concreto se escribe *X'hh'*.
- Cuando un campo se describe como *UInt32* se codifica en cuatro octetos, en big-endian.
- Los campos textuales se codifican siempre como secuencias UTF-8 sin byte NULL final. Se denotan como *TEXTO*
- En particular, la forma *TEXTO “usuario:contraseña” r*epresenta los bytes de la cadena concatenada, incluyendo el carácter : (X'3A').
- <paginado> significa que se maneja una respuesta extensa, lo cual está explicado en la sección *1.3.1.2 Manejo de Respuestas Extensas (Paginación).*
- Se utiliza ‘+’ para denotar que un valor está a continuación de otro. No se envía en el protocolo.

1.3.1.1 Formato de mensaje

Encabezado fijo de 3 octetos seguido de un PAYLOAD opcional:

Octeto	0	1	2	3…39
Campo	VER	CMD/STATUS	LEN	PAYLOAD
Descripción	Versión (X'01')	Comando (cliente) o estado (servidor)	Longitud del PAYLOAD (0-255)	Datos dependientes del comando

- **VER:** La versión del protocolo, fijada en X'01'.
- **CMD / STATUS:** En los mensajes del cliente, este campo contiene el código del **Comando** a ejecutar. En las respuestas del servidor, contiene el **Código de Estado** de la operación.
- **LEN:** Un octeto que indica la longitud en bytes del payload. Si es 0, el campo payload se omite.
- **PAYLOAD:** Los datos asociados al comando o la respuesta. Los campos textuales se codifican como secuencias US-ASCII sin un byte NULL terminal.

1.3.1.2 Manejo de Respuestas Extensas (Paginación)

Para comandos que pueden devolver una gran cantidad de datos (como listar usuarios o logs), el protocolo implementa un mecanismo de paginación o "chunking":

1. El cliente inicia la solicitud con un offset de 4 bytes (inicialmente 0).
2. El servidor responde con un PAYLOAD que comienza con el next_offset (4 bytes). Si este valor es 0, significa que no hay más datos. Si es distinto de cero, indica el punto de partida para la siguiente solicitud.
3. El cliente continúa pidiendo chunks con el next_offset recibido hasta que este sea 0.

Este sistema asegura que nunca se exceda el límite de 255 bytes por mensaje, sin importar el volumen total de los datos.

1.3.1.3 Códigos de estado (STATUS)

STATUS	Significado
X'00'	Operación exitosa
X'01'	Error genérico o parámetro inválido
X'02'	Autenticación requerida
X'03'	Credenciales inválidas
X'04'	Usuario no encontrado
X'05'	Tabla de usuarios llena
X’06’	Formato de payload invalido
X’07’	Usuario ya existe
X’08’	Tamaño de buffer no permitido
X’09’	Nombre de usuario reservado

1.3.1.2 Comandos

CMD	Descripción	PAYLOAD hacia servidor	Respuesta (STATUS)	Notas sobre el PAYLOAD de respuesta (si STATUS es X'00')
X'01'	Autenticación	TEXTO “usuario:contraseña”	X'00' | X'03'	
X'02'	Estadísticas		X'00'	UInt32 + UInt32 + UInt32 + UInt32 + UInt32 (aclaración 1)
X'03'	Listar usuarios	UInt32 (offset)	X’00’	<paginado> UInt32 + TEXTO (aclaración 2)
X'04'	Agregar usuario	TEXTO "usuario:contraseña"	X'00' | X'01'| X'05' | X’06’ | X’07 |, X’09’	
X'05'	Eliminar usuario	TEXTO "usuario"	X'00' | X'04'	
X'06'	Cambiar contraseña	TEXTO "usuario:nuevaClave"	X'00' | X'04' | X’06’	
X'07'	Establecer tamaño de buffer	UInt32 (tamaño en bytes) (aclaración 3)	X'00' | X'01' | X’06’ | X’08’	
X'08'	Consultar tamaño actual del buffer		X’00’	UInt32 (tamaño actual del buffer)
X'09'	Establecer método de autentificación del servidor SOCKS5	X'00' | X'01' (aclaración 4)	X'00' | X'01' | X’06’	
X'0A'	Consultar método		X'00'	X'00 '| X'01' (aclaración 3)
X'0B'	Obtener log	TEXTO "usuario" + X’00’ + UInt32 (offset)	X'00'	<paginado> UInt32 + TEXTO (aclaración 5)

Aclaraciones:

1. Los 5 valores UInt32 representan lo siguiente, en el respectivo orden: conexiones históricas, Conexiones actuales, bytes cliente al servidor, servidor al cliente
2. El UInt32 representa el next_offset de la paginación. Luego se encuentra una lista de usuarios, cada uno terminado en X’00’.
3. En CMD X’07’ solo se admiten los valores 0x00001000 (4,096), 0x00002000 (8,192), 0x00004000 (16,384),0x00008000 (32,768), 0x00010000 (65,536) y 0x00020000 (131,072) en el payload hacia el servidor.
4. En CMD X’09’ y X’0A’, en los payloads, el X'00' representa sin autentificación y X'01' representa con autenticación
5. El UInt32 representa el next_offset de la paginación. Luego están logs en texto, con un formato elegido por el servidor.

1.3.1.3 Ciclo de conexión

- Conexión TCP.
- Cliente envía CMD X'01' con credenciales (ASCII).
- El Servidor responde STATUS X'00' (éxito) o X'03' (falla).
- Con STATUS X'00' el cliente puede usar CMD X'02'–X'0B'; de lo contrario recibe STATUS X'02' cuando intenta usar uno de esos comandos sin estar autentificado.
- Cada comando válido genera una única respuesta.
- Cierre por FIN/RST o timeout.

1.3.1.4 Restricciones y validaciones generales

- **Validación de Longitud:** El servidor valida que el campo LEN coincida con los bytes realmente recibidos en el PAYLOAD. Una discrepancia resulta en el cierre de la conexión para prevenir ataques.
- **Manejo de Payloads Extensos:** El protocolo limita el PAYLOAD de un solo mensaje a 255 octetos. Para comandos que devuelven datos extensos (como la lista de usuarios o los logs), en lugar de truncar la información, se utiliza un **mecanismo de paginación** que divide la respuesta en múltiples "chunks" o trozos, asegurando la integridad de los datos.
- **Límites del Servidor:** El servidor tiene un límite configurable de **10 usuarios**. Si se intenta agregar más, responderá con STATUS X'05' (Tabla llena).

1.4 Servidor SOCKS5

El servidor SOCKS5 es la aplicación principal que implementa el protocolo SOCKS5 y el protocolo de management. Está diseñado como una aplicación de alto rendimiento que puede manejar múltiples conexiones simultáneas utilizando un patrón de máquina de estados. Cada conexión cliente pasa por diferentes estados: negociación inicial, autenticación (si es requerida), procesamiento de solicitudes, resolución de direcciones, establecimiento de conexiones a destinos y transferencia de datos.

El servidor incluye capacidades avanzadas como logging detallado de todas las conexiones en formato tabular, detección automática de credenciales en tráfico HTTP y POP3, estadísticas en tiempo real, y configuración dinámica de parámetros. Utiliza buffers eficientes para el manejo de datos y un selector I/O no bloqueante para maximizar el rendimiento con múltiples conexiones concurrentes.

1.5 Cliente SOCKS5

El cliente SOCKS5 es una aplicación interactiva desarrollada para probar y validar el funcionamiento del servidor proxy. Permite conectarse al servidor SOCKS5 y realizar solicitudes de conexión a destinos remotos. El cliente soporta tanto el modo autenticado como el no autenticado, y permite cambiar dinámicamente entre estos modos durante la sesión.

La aplicación incluye funcionalidades para realizar pruebas HTTP básicas, solicitudes personalizadas a destinos específicos, y soporte completo para IPv4, IPv6 y nombres de dominio. El cliente maneja automáticamente los timeouts de conexión y proporciona retroalimentación detallada sobre el estado de las operaciones, incluyendo mensajes de error específicos cuando las conexiones fallan.

1.6 Cliente de Management

El cliente de management es una herramienta de administración que permite interactuar con el protocolo de management del servidor. Requiere autenticación obligatoria como administrador antes de permitir cualquier operación. Una vez autenticado, proporciona una interfaz interactiva para consultar estadísticas del servidor, gestionar usuarios autorizados y modificar configuraciones del sistema.

El cliente incluye comandos para listar usuarios existentes, agregar nuevos usuarios con sus contraseñas, eliminar usuarios del sistema, cambiar contraseñas de usuarios existentes, consultar estadísticas de conexiones y bytes transferidos, y modificar configuraciones como el tamaño de buffers o el método de autenticación

global. Todas las operaciones se comunican con el servidor utilizando el protocolo de management y proporcionan retroalimentación inmediata sobre el éxito o fracaso de cada operación.

1.7 Componentes de Soporte

El sistema incluye varios componentes de soporte que proporcionan funcionalidades comunes utilizadas por todas las aplicaciones. El sistema de buffers implementa buffers circulares eficientes para el manejo de datos de entrada y salida, con operaciones optimizadas de lectura y escritura. El selector I/O proporciona multiplexación no bloqueante de múltiples conexiones, permitiendo que el servidor maneje eficientemente cientos de conexiones simultáneas.

La máquina de estados es un motor genérico que maneja las transiciones entre diferentes estados de las conexiones, permitiendo una implementación limpia y mantenible del protocolo SOCKS5. El sistema de logging proporciona diferentes niveles de registro (error, advertencia, información, debug) y incluye funcionalidades especializadas para registrar acceso y detectar credenciales en el tráfico de red. Todos estos componentes están diseñados para ser reutilizables y eficientes, proporcionando una base sólida para el desarrollo de aplicaciones de red de alto rendimiento.

2. Problemas encontrados durante el diseño y la implementación

2.1 Resolución DNS Asíncrona con getaddrinfo_a()

Uno de los desafíos más significativos fue la implementación de la resolución DNS asíncrona utilizando getaddrinfo_a(). Esta función no está disponible en todos los sistemas operativos y nos enfrentamos a errores de compilación dependiendo de la plataforma de desarrollo. La función requiere la librería libanl en Linux y puede no estar presente en otros sistemas Unix.
Implementamos un sistema híbrido que utiliza getaddrinfo_a() para resolver las consultas DNS, y creamos estructuras addrinfo manualmente para el caso de recibir direcciones IP directas. Esto requirió un manejo cuidadoso de la memoria, ya que getaddrinfo_a() utiliza freeaddrinfo() mientras que las estructuras manuales requieren liberación individual de cada componente.

2.2 Limitaciones de IPv6 en Entornos de Desarrollo

La implementación de IPv6 se vio limitada por las restricciones de nuestros entornos de desarrollo. Los routers y redes locales no proporcionaban conectividad IPv6 real, lo que dificultó las pruebas exhaustivas de esta funcionalidad. Aunque el código incluye soporte completo para IPv6 (incluyendo estructuras sockaddr_in6 y manejo de direcciones IPv6), no pudimos validar con una conexión que no sea en nuestra red local el comportamiento en escenarios reales de red IPv6.

2.3 Manejo de Memoria Dinámica y Recursos

El manejo de la memoria dinámica fue un aspecto crítico, especialmente al gestionar buffers de tamaño variable y estructuras asociadas a cada conexión. Fue necesario implementar rutinas cuidadosas para liberar correctamente la memoria de buffers, estructuras de usuarios y resultados de resoluciones DNS, evitando fugas de memoria y corrupción de datos. Además, el manejo de file descriptors (FDs) requirió especial atención para evitar leaks: cada conexión cliente y cada conexión a destino abren FDs que deben ser cerrados correctamente en todos los caminos de error y cierre, para no agotar los recursos del sistema.

2.4 Manejo de Estados Erróneos y Robustez de la Máquina de Estados

El diseño de la máquina de estados implicó prever y manejar transiciones erróneas o inesperadas. Fue necesario implementar validaciones y retornos de error en cada estado para evitar que una conexión quedará “colgada” o en un estado inconsistente ante errores de red, fallos de autenticación, o problemas internos. La robustez de la máquina de estados es clave para la estabilidad del servidor, especialmente bajo carga o ante clientes maliciosos.

2.5 Protección ante Ataques de Denegación de Servicio (DoS) en el Parser

Durante el desarrollo, identificamos que los parsers (tanto de SOCKS5 como de management) pueden ser un vector de ataque para intentos de denegación de servicio (DoS), por ejemplo, enviando mensajes malformados, muy grandes o secuencias de bytes que nunca completan un mensaje válido. Para mitigar esto, se implementaron límites estrictos en los tamaños de payload, validaciones de longitud y chequeos de integridad en cada paso del parsing. Además, se implementó un timeout para cada fd en el cual si detecta que está inactivo esperando en el parseo por más de 60 seg el mismo corta ejecución.

3. Limitaciones de la aplicación

La aplicación, si bien es robusta y funcional, presenta algunas limitaciones importantes. En primer lugar, el servidor SOCKS5 implementa únicamente el comando CONNECT, por lo que no es posible utilizarlo para tráfico UDP ni para escenarios de reverse proxy, limitando su uso a conexiones TCP salientes. Además, el sistema de gestión de usuarios es básico: permite un máximo de 10 usuarios, con credenciales almacenadas solo en memoria RAM, lo que implica que cualquier reinicio del servidor borra los usuarios agregados dinámicamente y no existe persistencia de configuración ni de estadísticas.
Por otro lado, aunque el código incluye soporte para IPv6, no fue posible realizar pruebas exhaustivas debido a la falta de conectividad real en los entornos de desarrollo(igualmente hemos podido hacer algunas pruebas con lo que nos respondió Thomas en las consultas) , por lo que pueden existir casos límite no detectados. La aplicación también depende de funciones POSIX específicas, como getaddrinfo_a() y el manejo de señales, lo que puede dificultar su portabilidad a otros sistemas operativos y requiere ajustes para compilar en diferentes entornos Unix/Linux.
Finalmente, la falta de persistencia, la ausencia de cifrado en las comunicaciones y la gestión de recursos limitada restringen su uso en ambientes de producción o de alta seguridad. Estas limitaciones son importantes a considerar para futuras mejoras o despliegues en escenarios reales.

4. Posibles extensiones

El sistema desarrollado ofrece una base sólida y extensible, sobre la cual se pueden implementar numerosas mejoras y nuevas funcionalidades para ampliar su alcance y robustez. Algunas posibles extensiones incluyen:
Una de las mejoras más directas sería implementar el soporte para los comandos BIND y UDP ASSOCIATE del protocolo SOCKS5, permitiendo así el manejo de tráfico UDP y escenarios de reverse proxy, lo que haría al servidor compatible con la totalidad del estándar y útil para una gama más amplia de aplicaciones, como juegos en línea o servicios de VoIP.
Otra extensión relevante sería la persistencia de usuarios y configuración, permitiendo almacenar en disco la lista de usuarios, parámetros de configuración y estadísticas, de modo que sobrevivan a reinicios del servidor. Esto podría complementarse con la posibilidad de cargar y guardar configuraciones desde archivos o bases de datos.
En términos de seguridad, sería deseable agregar soporte para cifrado en las comunicaciones, tanto en el canal SOCKS5 como en el de management, por

ejemplo mediante TLS/SSL. Además, se podrían implementar mecanismos de protección contra ataques de fuerza bruta (como bloqueo temporal de usuarios tras varios intentos fallidos) y controles de acceso más granulares.

5. Ejemplos de prueba

Utilización del proxy en firefox: Se configuró firefox para la utilización del proxy dentro del mismo navegador de esta manera pudimos probar de manera visual la velocidad del mismo y cómo reaccionaba antes la apertura de múltiples pestañas en sitios como youtube.

Testeo con script de bash: Se programó un script de bash que hace 500 curls utilizando el puerto 81 de google ya que este puerto no responde y se queda en EINPROGRESS el intento de connect por lo que nos sirvió para probar las 500 conexiones simultáneas en un mismo estado. también concluimos que el máximo de usuario concurrentes que intentan conectarse es de 509.

```
#!/bin/bash
for i in {1..500}; do
    curl -x socks5h://127.0.0.1:1080 http://www.google.com:81
--max-time 5 > /dev/null 2>&1 &
done

wait
```

Testeo de velocidad con diferentes tamaños de buffer:

Para probar la velocidad con los distintos buffers levantamos el servidor SOCKS5 en pampero y, en la misma máquina, un servidor HTTP simple (python -m http.server) que alojó un archivo de 3,7 GB. Desde otra computadora se descargó el archivo vía curl, forzando todo el tráfico a pasar por el proxy (--socks5), y se midió el tiempo total con time. El único parámetro que se modificó entre ensayos fue el tamaño del buffer interno del proxy (4 KB, 8 KB y 128 KB).

Los resultados muestran una mejora clara al aumentar el buffer: con 4 KB la descarga tomó unos 16 min (≈3,8 MB/s); con 8 KB bajó a 11 min (≈5,8 MB/s); y con 128 KB quedaron apenas 8 min (≈8 MB/s). La ganancia proviene de **reducir la cantidad de llamadas al sistema por segundo, de modo que la CPU dedica más tiempo a mover datos y menos a gestionar interrupciones**.

En síntesis, para transferencias grandes a través de este proxy conviene usar buffers de al menos 64–128 KB: se aprovecha mejor el ancho de banda disponible sin un consumo de memoria significativo.

```
sangui1:~ valentinosanguinetti$ time curl --socks5 200.5.121.137:1080 -L -k -o /dev/null "http://200.5.121.137:8082/test_5gb.bin"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3705M  100 3705M    0     0  3863k      0  0:16:22  0:16:22 --:--:-- 7735k

real    16m22.086s
user    0m9.900s
sys     0m39.102s
```

FIG: prueba con buffer de 4096 bytes.

```
sangui1:~ valentinosanguinetti$ time curl --socks5 200.5.121.137:1080 -L -k -o /dev/null "http://200.5.121.137:8082/test_5gb.bin"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3705M  100 3705M    0     0  5795k      0  0:10:54  0:10:54 --:--:-- 9175k

real    10m54.669s
user    0m9.498s
sys     0m37.053s
```

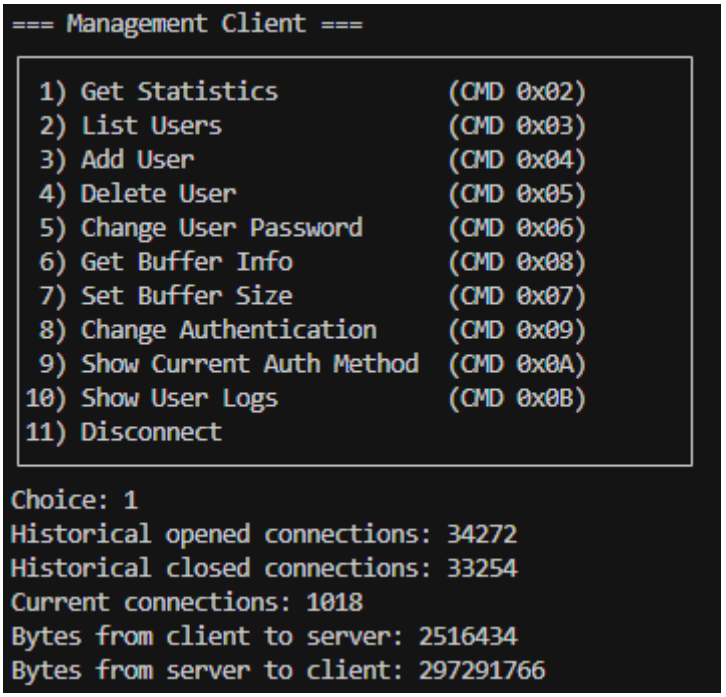
FIG: prueba con buffer de 8192 bytes.

```
sangui1:~ valentinosanguinetti$ time curl --socks5 200.5.121.137:1080 -L -k -o /dev/null "http://200.5.121.137:8082/test_5gb.bin"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3705M  100 3705M    0     0  7963k      0  0:07:56  0:07:56 --:--:--  9.9M

real    7m56.450s
user    0m9.450s
sys     0m38.894s
```

FIG: prueba con buffer de 131072 bytes.

Testeo de conexiones concurrentes:
Luego también hicimos distintas pruebas para ver efectivamente cuantas conexiones simultaneas soporta nuestro servidor. Dejamos en la carpeta el test3.sh el cual se encarga de ir generando conexiones y dejarlas abiertas. A medida que iba pasando eso podíamos ver desde el cliente de management cuantas conexiones había efectivamente al mismo tiempo conectadas. Nos dimos cuenta que el máximo de conexiones concurrentes fue de 1018. Estuvimos viendo y repasando el código y entendimos que esto pasa porque el límite máximo de conexiones simultáneas que soporta el servidor está dado por la arquitectura basada en select() y la constante FD_SETSIZE del sistema operativo, que en Linux es típicamente 1024. Como el servidor utiliza algunos descriptores para los sockets de escucha y otros recursos, el número real de conexiones de clientes suele ser un poco más pequeño a 1024 (en nuestras pruebas, 1018). Esto solo pasa cuando las conexiones no intentan conectarse a ningún lado, osea se conecta al servidor mediante netcat, ya que si se intentara conectar las conexiones se reducirán a la mitad ya que por cada conexión se necesitan 2 sockets.



Por último hicimos un test para medir el throughput:
A medida que se incrementa la cantidad de conexiones simultáneas, el throughput del servidor empieza a degradarse. Esto se debe a que el multiplexor basado en select() debe iterar sobre todos los FD(File Descriptors) en cada ciclo, lo que incrementa el tiempo de procesamiento por evento. En nuestras pruebas, observamos que el throughput se mantiene relativamente estable hasta aproximadamente 300-600 conexiones, pero a partir de allí comienza a disminuir de manera significativa, acompañado de un aumento en la latencia y en la tasa de errores. Al acercarse al límite impuesto por FD_SETSIZE, el servidor experimenta una degradación abrupta del rendimiento, llegando incluso a rechazar nuevas conexiones o a presentar timeouts frecuentes.

6. Guia de instalacion

Para la compilación y ejecución del programa se necesita tener **GCC** y **Make** en sus versiones más recientes.
Para compilar los binarios se pueden correr los siguientes dos comandos:

```
make all
make all BUILD=debug
```

El primer comando sirve para compilar una versión más rápida del código ignorando flags de chequeo de memoria que puede ralentizar el código, además de cambiar el nivel de los logs ya que los debug no se imprimen. El segundo comando sirve para hacer una validación del código y entender bien qué está pasando ante cualquier fallo del programa.
Para limpiar el proyecto se debe correr el siguiente comando.

```
make clean
```

Servidor proxy socksv5

Para ejecutar el proxy se debe correr el siguiente comando.

```
./bin/server/socks5d
```

agregar el flag -h imprime los diferentes flags que acepta y sus argumentos. Es posible definir las variables de entorno *ADMIN_USERNAME* y *ADMIN_PASSWORD* antes de ejecutar el servidor para cambiar las credenciales de administrador para autenticarse dentro del protocolo management.

Cliente de management

Para ejecutar el cliente de management se debe correr el siguiente comando.

```
./bin/server/mgmt_client
```

En cuanto a la autenticación, si se definieron las variables de entorno mencionadas anteriormente, se deben utilizar aquellos valores. En el caso de que no se hayan definido, se usan los valores por defecto, los cuales con admin para el usuario y admin123 para la contraseña.

Cliente de proxy

Para ejecutar el cliente de proxy se debe correr el siguiente comando.

```
./bin/server/socks5_client
```

7. Instrucciones de Configuración

Setear usuario admin:

El sistema de administración del servidor permite la autenticación de un único usuario administrador, cuyas credenciales pueden configurarse de dos maneras: una es cambiando desde el código fuente las que pusimos por default (admin:admin123), o bien estableciendo las variables de entorno ADMIN_USERNAME y ADMIN_PASSWORD antes de iniciar el servidor.

```
valen@MSI:/mnt/c/users/tpi/documents/github/tp-protos/bin/server$ export ADMIN_USERNAME=miusuario
valen@MSI:/mnt/c/users/tpi/documents/github/tp-protos/bin/server$ export ADMIN_PASSWORD=clave
valen@MSI:/mnt/c/users/tpi/documents/github/tp-protos/bin/server$ ./socks5d
```

De esta forma, el administrador puede personalizar el nombre de usuario y la contraseña sin necesidad de modificar el código, simplemente configurando dichas variables en el entorno de ejecución. Esta autenticación es requerida para acceder a las funciones de gestión y control del servidor a través del protocolo de administración.

Setear direcciones y puertos para socks5 y management:

Al correr el servidor se pueden usar los siguientes flags para setear los puertos y direcciones:

- -l <SOCKS addr>: Dirección donde servirá el proxy SOCKS (default: 0.0.0.0)
- -L <conf addr>: Dirección donde servirá el servicio de management (default: 127.0.0.1)
- -p <SOCKS port> - Puerto entrante conexiones SOCKS (default: 1080)
- -P <conf port> - Puerto entrante conexiones configuración/management (default: 8080)

8. Ejemplos de configuración y monitoreo

Monitoreo:

Conexión al servidor mediante el cliente de management:

```
fer@fer-ThinkPad-T14s-Gen-2i:~/CLionProjects/tp-protos$ ./bin/client/mgmt_client
Management server IP/domain [127.0.0.1]:
Management server port [8080]: 8081
1) Connect & Authenticate
2) Exit
Choice: 1
Successfully connected to 127.0.0.1:8081
Username: admin
Password: admin123
Success

=== Management Client ===

 1) Get Statistics           (CMD 0x02)
 2) List Users               (CMD 0x03)
 3) Add User                 (CMD 0x04)
 4) Delete User              (CMD 0x05)
 5) Change User Password     (CMD 0x06)
 6) Get Buffer Info          (CMD 0x08)
 7) Set Buffer Size          (CMD 0x07)
 8) Change Authentication    (CMD 0x09)
 9) Show Current Auth Method (CMD 0x0A)
10) Show User Logs           (CMD 0x0B)
11) Disconnect
```

Consultar estadísticas:

```
Choice: 1
Historical opened connections: 204
Historical closed connections: 204
Current connections: 0
Bytes from client to server: 1306553
Bytes from server to client: 77402629
```

Listar usuarios:

```
Choice: 2
Users:
- admin
- user1
- user2
--- End of list (3 users) ---
```

Añadir usuario (y comprobar listando):

```
Choice: 3
New user: marce
New pass: garbe
Success

=== Management Client ===

 1) Get Statistics            (CMD 0x02)
 2) List Users                (CMD 0x03)
 3) Add User                  (CMD 0x04)
 4) Delete User               (CMD 0x05)
 5) Change User Password      (CMD 0x06)
 6) Get Buffer Info           (CMD 0x08)
 7) Set Buffer Size           (CMD 0x07)
 8) Change Authentication     (CMD 0x09)
 9) Show Current Auth Method  (CMD 0x0A)
10) Show User Logs            (CMD 0x0B)
11) Disconnect

Choice: 2
Users:
- admin
- user1
- user2
- marce
--- End of list (4 users) ---
```

Cambiar contraseña de usuario:

```
Choice: 5
User: marce
New pass: pass
Success
```

Consultar tamaño de buffer:

```
Choice: 6
Current buffer size: 32768 bytes
```

Cambiar tamaño del buffer:

```
Choice: 7
Buffer sizes:
 1) 4096
 2) 8192
 3) 16384
 4) 32768
 5) 65536
 6) 131072
Choice: 5
Success
```

Cambiar el método de autenticación
(en este caso para aceptar conexiones sin autenticar):

```
Choice: 8
1) NOAUTH
2) AUTH
Choice: 1
Success
```

Consultar método de autenticación actual:

```
Choice: 9
NOAUTH
```

Mostrar logs de un usuario:

```
Choice: 10
User ("anonymous" for NOAUTH): marce

--- Logs for marce ---
2025-07-15T12:27:33Z    127.0.0.1       54484   142.251.129.142 80      0x00
2025-07-15T12:27:58Z    127.0.0.1       57554   google.com      80      0x00
--- End of logs ---
```

Desconectarse:

```
Choice: 11
1) Connect & Authenticate
2) Exit
Choice: 2
fer@fer-ThinkPad-T14s-Gen-2i:~/CLionProjects/tp-protos$
```

Configuración:

Setear puertos para socks5 y management:

En este ejemplo seteamos socks5 en el puerto 1081 y management en el 8081. Además agregamos un usuario inicial con nombre de usuario “user” y contraseña “pass”.

```
fer@fer-ThinkPad-T14s-Gen-2i:~/CLionProjects/tp-protos$ ./bin/server/socks5d -p 1081 -P 8081 -u user:pass
```

Setear credenciales del admin:

En este ejemplo seteamos el usuario (miusuario) y contraseña (clave) del administrador haciendo uso de las variables de entorno.

```
valen@MSI:/mnt/c/users/tpi/documents/github/tp-protos/bin/server$ export ADMIN_USERNAME=miusuario
valen@MSI:/mnt/c/users/tpi/documents/github/tp-protos/bin/server$ export ADMIN_PASSWORD=clave
valen@MSI:/mnt/c/users/tpi/documents/github/tp-protos/bin/server$ ./socks5d
```

9. Documento de diseño del proyecto

La estructura del proyecto es la siguiente:

- **1. Sistema de Compilación y Documentación Raíz**
  - *Makefile*: Compilación de todo el proyecto.
  - *README.md*: Documentación principal.
  - *socks5d.8*: Página de manual del servidor.
  - *test.sh, test2.sh*: Scripts de prueba.
- **2. Código Fuente (src)**
  - **2.1. Aplicaciones Cliente (src/Client)**
    - *mgmt_client.c:* Cliente para el protocolo de gestión.
    - *socks5_client.c*: Cliente de prueba para el SOCKS5.
    - *client_utils.c, client_utils.h*: Funciones de utilidad para los clientes.
  - **2.2. Lógica del Servidor (src/Server)**
    - **2.2.1. Punto de Entrada y Configuración**
      - *main.c*: Inicio del servidor, configuración de sockets y bucle de eventos principal.
      - *args.c, args.h*: Parseo de argumentos de línea de comandos.
    - **2.2.2. Núcleo**
      - *selector.c, selector.h*: Multiplexor de I/O no bloqueante.
      - *stm.c, stm.h*: Motor de máquina de estados.
      - *buffer.c, buffer.h*: Implementación de buffer.
    - **2.2.3. Lógica de Protocolos**
      - **Protocolo SOCKS5**:
        - *sock5.c, sock5.h*: Orquestador principal de las conexiones SOCKS5.
        - *Negotiation/*: Lógica para la etapa de negociación.
        - *Auth/*: Lógica para la etapa de autenticación.
        - *Resolver/*: Lógica para la etapa de solicitud y conexión.
        - *Copy/*: Lógica para la etapa de copiado de datos.
      - **Protocolo Management**:
        - *ManagementProtocol/*: Lógica para el protocolo de administración.
    - **2.2.4. Módulos de Soporte del Servidor**
      - *Statistics/*: Lógica para la recolección de métricas.
      - *users.h*: Definiciones de la estructura de usuarios del servidor.
      - *constants.h, protocol_constants.h*: Constantes específicas del servidor.
    - **2.2.5. Pruebas Unitarias (src/Server/Tests)**
      - Pruebas individuales para los componentes del núcleo.
  - **2.3. Archivos Compartidos (src/)**
    - *logger.h*: Interfaz del sistema de logging global.
    - *management_constants.h*: Constantes compartidas entre el cliente y servidor de gestión.
    - *Makefile.inc*: Incluido por otros Makefiles para la compilación.

La arquitectura del servidor (2.2 src/Server) se basa en un diseño **monohilo y dirigido por eventos** para manejar un gran número de conexiones concurrentes sin el overhead de crear un hilo por cada una. Se sustenta sobre los componentes del **núcleo** (2.2.2). El **Selector** notifica cuándo un socket está listo para la operación indicada, y el motor de **Máquina de Estados (STM)** gestiona la lógica de cada conexión individual. Para gestionar los flujos de datos de manera eficiente, este modelo se apoya en una estructura de **Buffer** de acceso directo.

En main.c (2.2.1) registramos los sockets pasivos en el selector. Estos escucharán los pedidos de conexión para ambos protocolos.

En sock5.c y sock5.h (2.2.3) definimos los estados posibles para una conexión SOCKS5 y creamos los handlers correspondientes para cada estado y operación. El ciclo de vida de una conexión se divide en las siguientes etapas, cada una con sus propios estados y con su lógica encapsulada en un subdirectorio dedicado:

- **Etapa de Negociación**: Se encarga del handshake inicial para acordar el método de autenticación. Consiste de los estados NEGOTIATION_READ, NEGOTIATION_WRITE y las funciones asociadas a estos estados se encuentran en src/Server/Negotiation/.
- **Etapa de autenticación**: Valida las credenciales del usuario si el método acordado lo requiere. Consiste de los estados AUTHENTICATION_READ, AUTHENTICATION_WRITE, AUTHENTICATION_FAILURE_WRITE y las funciones de dichos estados se encuentran en src/Server/Auth/.
- **Etapa de Solicitud y Conexión**: Parsea la petición del cliente, resuelve el nombre de dominio de forma asíncrona si es necesario, y establece la conexión con el servidor de destino de forma no bloqueante. Tiene los estados REQ_READ, ADDR_RESOLVE, CONNECTING, REQ_WRITE. El código se ubica en src/Server/Resolver/.
- **Etapa de Copia de Datos**: Una vez establecida la conexión, entra en este estado final donde actúa como un "pipe", transfiriendo datos de forma bidireccional entre el cliente y el destino. El estado es COPYING y el código se ubica en src/Server/Copy/.

De forma paralela al servicio SOCKS5, el servidor expone el protocolo **management** el cual reutiliza la misma arquitectura utilizando el motor de máquina de estados y el selector. Todo el código se encuentra en src/Server/ManagementProtocol y su flujo se divide en dos fases principales:

- **1. Fase de Autenticación**: Toda conexión debe primero autenticarse. Este paso se maneja a través de los estados MGMT_AUTH_READ y MGMT_AUTH_WRITE.
- **2. Fase de Comandos**: Una vez autenticado, el cliente entra en un bucle de lectura-escritura para procesar los comandos del administrador. Este ciclo se gestiona con los estados MGMT_COMMAND_READ y MGMT_COMMAND_WRITE. La ejecución de cada comando se delega a funciones específicas a través de una tabla de punteros a función en management_cmds.c.

Los estados terminales para esta máquina son MGMT_CLOSED y MGMT_ERROR

10. Conclusiones

Llevar a cabo la implementación de un servidor SOCKS5 y sus clientes fue todo un desafío: desde pensar el protocolo y optimizar los recursos hasta construir las pantallas de administración y de monitoreo en tiempo real. Finalmente, pudimos crear una solución modular y extensible, que soportaba múltiples conexiones concurrentes sin problemas.

Un punto clave en la mejora del rendimiento fue gracias a la corrección que nos hizo Sebas en el simulacro. Originalmente, el servidor seguía la secuencia select → read → select → write, pero Sebas nos sugirió optimizarlo a select → read → write, verificando que el write puede devolver EWOULDBLOCK. Esta modificación nos permitió reducir la cantidad de ciclos de select innecesarios y aprovechar mejor cada notificación de disponibilidad, mejorando notablemente el performance de la aplicación y ayudándonos a entender por qué era más eficiente hacerlo de esa manera.

Todo esto fue posible gracias a lo aprendido en la cursada. Conceptos que quizás al principio no entendíamos del todo se terminaron transformando en herramientas super necesarias para ir avanzando con el proyecto. El foro tuvo un rol fundamental: a partir de consultas y respuestas sumadas a los consejos de los docentes como de los compañeros, pudimos superar muchos de los problemas con los que nos fuimos encontrando.

Lo mejor fue ver cómo llegamos a un servidor que contaba con funciones avanzadas y con un nivel de robustez que nunca imaginamos al principio, partiendo de cero. Además de mejorar en el área de programación de protocolos, el proyecto nos enseñó el valor de un óptimo diseño, una buena documentación y flexibilidad para adaptarnos a las contingencias.

En definitiva, una experiencia que nos resulta muy enriquecedora y que nos deja mejor preparados para encarar proyectos aún más complejos en el futuro.