Trabajo Práctico Especial - Protocolos de Comunicación

Grupo 5

Índice

Integrantes

Apellido	Nombre	Legajo	E-mail
Borinsky	Lucila	63039	lborinsky@itba.edu.ar
Diaz Sieiro	Santiago	63058	sdiazsieiro@itba.edu.ar
Percich	Luana	64316	lpercich@itba.edu.ar
Trajterman	Catalina	64175	ctrajterman@itba.edu.ar

Protocolos y aplicaciones desarrolladas

El objetivo principal de este Trabajo Práctico Especial fue desarrollar un servidor proxy que implemente el protocolo SOCKS5 conforme a las especificaciones del RFC 1928 y sus mecanismos asociados de autenticación. Para esto, se utilizó una arquitectura basada en una máquina de estados y un selector capaz de gestionar conexiones simultáneas sin bloquearse.

El proxy comienza realizando el *handshake* correspondiente y negociando el método de autenticación con el cliente. En caso de utilizarse el método de usuario y contraseña, las credenciales son validadas contra la estructura interna de usuarios del sistema. Una vez superada esta etapa, el servidor se encarga de interpretar la *request* SOCKS5, resolviendo direcciones IPv4, IPv6 o dominios mediante un sistema DNS no bloqueante. Luego de conectarse con el destino solicitado, se responde al cliente con la *reply* correspondiente. Posteriormente, el servidor pasa al modo de *relay*, donde reenvía datos entre el cliente y el servidor destino utilizando buffers independientes y operaciones no bloqueantes.

Por otro lado, además del protocolo SOCKS5, el trabajo incluyó la creación de un protocolo de administración, utilizando una interfaz por TCP que permite autenticación, consultar métricas (conexiones activas, conexiones históricas y bytes transferidos) y gestionar usuarios del proxy, ya sea agregando un nuevo usuario (brindando nombre de usuario y contraseña), eliminando un usuario a partir de su nombre o listando los usuarios existentes.

En conjunto, el trabajo desarrollado en C integra negociación, autenticación, parsing, resolución DNS, forwarding no bloqueante y administración remota, dando como resultado un proxy completo y conforme con las especificaciones del RFC mencionado.

Problemas encontrados

A lo largo del desarrollo del trabajo práctico, distintos obstáculos fueron encontrados en distintas etapas. La primera dificultad fue encontrada al notar que el ISP implementado no proveía una conectividad IPv6 nativa, lo que impedía probar el requerimiento funcional de conexiones de este tipo contra servidores reales en internet. Para solucionarlo, se decidió montar un entorno de pruebas sintético local, usando servidores HTTP en Python con bind a ::1 para validar que el parser y la lógica del socket funcionaran correctamente, simulando el escenario real.

Por otro lado, al verificar el funcionamiento de la máquina de estados con Firefox se encontró un Segmentation Fault recurrente. Esto se debía a que el selector disparaba eventos de lectura (**`OP_READ`**) mientras la conexión estaba esperando al DNS (en el estado **`REQUEST_RESOLVE`**). Que dicho estado no tuviera definido un

handler de lectura provocó que la máquina de estados explotara. Fue complejo diagnosticar que el problema ocurría por no “desregistrar” los intereses (**OP_NOOP**) con la suficiente rapidez al cambiar de estado. Como solución se optó por implementar handlers *dummy* de seguridad para evitar que eventos del selector tumbaran el servidor.

Limitaciones de la aplicación

A pesar de cumplir con los requerimientos del RFC y las funcionalidades principales del servidor proxy, la implementación provista presenta algunas limitaciones. En primer lugar, se cuenta con un soporte limitado de comandos SOCKS5, ya que solamente fue implementado el comando **CONNECT**. No fueron incluidos **BIND** ni **UDP ASSOCIATE**, lo que reduce el conjunto de funcionalidades. Además, si bien la autenticación de usuario/contraseña es funcional, no incluye cifrado ni mecanismos avanzados de seguridad. Por último, las métricas son volátiles, por lo que en caso de que se reinicie el servidor, las estadísticas pueden perderse.

Posibles extensiones

Si bien el servidor cumple con los objetivos principales, existen varias ideas que permitirían ampliar sus capacidades y volverlo más completo. Principalmente, en un futuro podrían implementarse los comandos **BIND** y **UDP ASSOCIATE** en el protocolo SOCKS5, extendiendo así el alcance del proyecto. Por otro lado, en el otro protocolo podrían implementarse más funciones para manejo de usuarios, como una

función que permita editar la información de un usuario (en un principio su contraseña) sin tener que eliminarlo y volverlo a añadir. De la misma manera, se podría hacer uso de métricas más avanzadas, para lograr una tabla de estadísticas más completa y un mejor monitoreo del funcionamiento. Por último, podría agregarse un soporte para cifrado en el canal de administración, empleando TLS para proteger credenciales de usuarios.

Conclusiones

En síntesis, el desarrollo de este servidor permitió integrar conceptos fundamentales tratados a lo largo de la cursada. Se considera que la implementación cumple con todos los requerimientos, a pesar de sus limitaciones. Por otro lado, la implementación junto con las limitaciones forman una base que brinda espacio a futuras extensiones. En conjunto, el proyecto resultó un servidor robusto y funcional, que permitió dar un cierre práctico a la materia.

Ejemplos de prueba

Para verificar el funcionamiento real del servidor proxy, se realizaron una serie de pruebas prácticas enfocadas en demostrar sus capacidades principales. El objetivo fue confirmar que el sistema puede conectarse tanto a sitios web tradicionales (IPv4) como a entornos de red más modernos (IPv6), y asegurar que el servidor responde correctamente ante las solicitudes de los clientes

- **Prueba de conectividad a Internet (IPv4):** Se comprobó que el proxy permite la navegación fluida hacia sitios web públicos. Utilizando la

herramienta **`curl`** como cliente, se realizó una petición a Google pasando a través del servidor desarrollado.

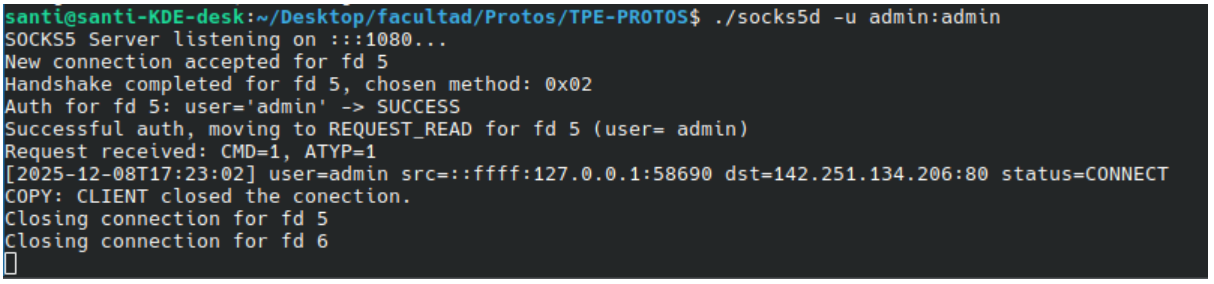
El comando ejecutado fue:

```
curl -v -x socks5://admin:admin@127.0.0.1:1080 142.251.134.206
```

Los resultados observados fueron:

Terminal que ejecutó el Curl:

Output del servidor imprimiendo la conexión exitosa:



Es importante notar que se utilizó el método de autenticación requerido para hacer dicha conexión a modo de ejemplo.

- **Validación de conexión IPv6 en entorno local:** Dadas las limitaciones de conectividad IPv6 nativa del proveedor de servicios de internet (ISP) disponible, se procedió a validar este requerimiento mediante un entorno

sintético local. Se creó un servidor web pequeño utilizando Python, configurado para escuchar exclusivamente en la dirección local de IPv6 (**::1**) en el puerto 8888 con el siguiente comando:

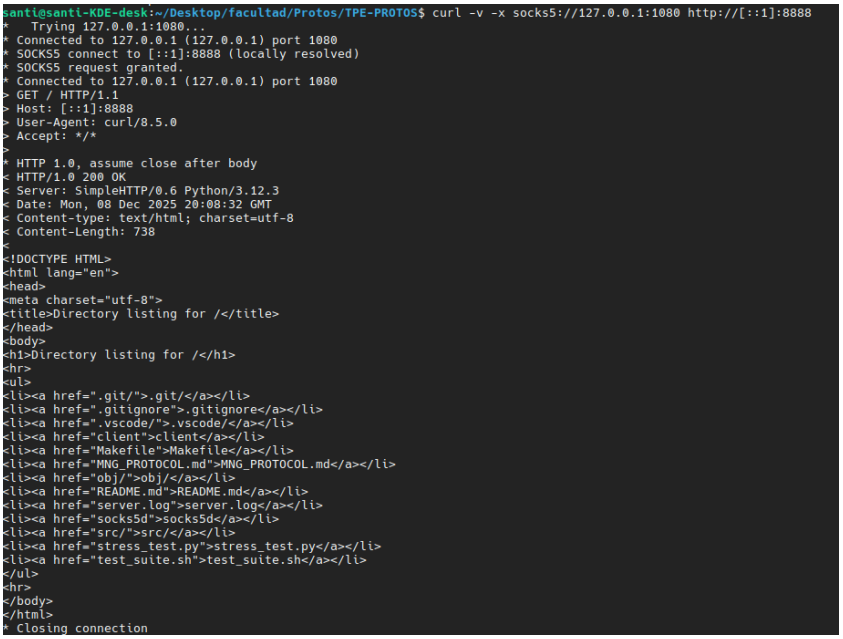
```
python3 -m  http.server -b ::1 8888 &
```

Para probar la conexión se empleó:

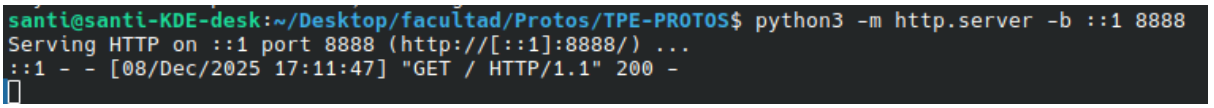
```
curl -v -x socks5://127.0.0.1:1080 http://[::1]:8888
```

Los resultados observados fueron:

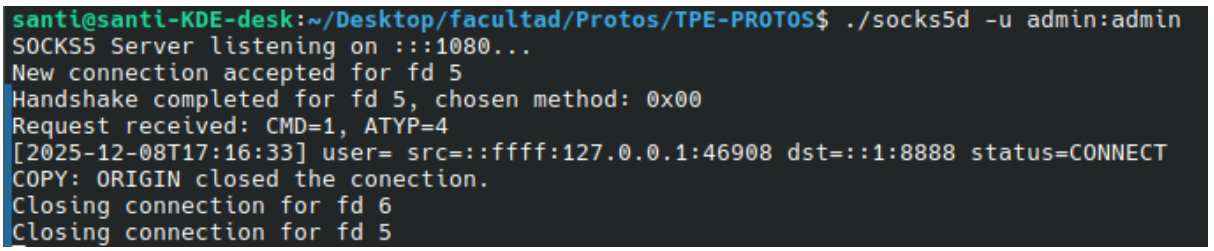
Terminal que ejecutó el curl:



Terminal que levantó el servidor IPv6:



Output del servidor imprimiendo la conexión exitosa:



El proxy logró interpretar la dirección IPv6 (ATYP 4), conectarse al servidor Python local y transmitir la respuesta al cliente. Esta prueba fue fundamental para verificar que el código está preparado para manejar el estándar IPv6, independientemente de las limitaciones de la conexión a internet externa.

- **Validación de resolución de nombres (DNS):** Se verificó la capacidad del servidor proxy para manejar la resolución de nombres de dominio (ATYP 3) delegada por el cliente. Esta funcionalidad es crítica para permitir que clientes sin capacidad de resolución DNS propia (o que deseen ocultar sus consultas DNS) naveguen a través del proxy. Se utilizó curl forzando la resolución remota (socks5h) para solicitar un dominio público (yahoo.com):

Terminal que ejecutó el curl:

```
redirectsanti@santi-KDE-desk:~/Desktop/facultad/Protos/TPE-PROTOS$ curl -v -x socks5h://127.0.0.1:1080 http://yahoo.com
*   Trying 127.0.0.1:1080...
* Connected to 127.0.0.1 (127.0.0.1) port 1080
* SOCKS5 connect to yahoo.com:80 (remotely resolved)
* SOCKS5 request granted.
* Connected to 127.0.0.1 (127.0.0.1) port 1080
> GET / HTTP/1.1
> Host: yahoo.com
> User-Agent: curl/8.5.0
> Accept: */*
>
< HTTP/1.1 301 Moved Permanently
< Date: Mon, 08 Dec 2025 20:33:35 GMT
< Connection: keep-alive
< Server: ATS
< Cache-Control: no-store, no-cache
< Content-Type: text/html
< Content-Language: en
< X-Frame-Options: SAMEORIGIN
< Location: https://yahoo.com/
< Content-Length: 8
<
* Connection #0 to host 127.0.0.1 left intact
```

Output del servidor imprimiendo la conexión exitosa:

```
santi@santi-KDE-desk:~/Desktop/facultad/Protos/TPE-PROTOS$ ./socks5d -u admin:admin
SOCKS5 Server listening on :::1080...
New connection accepted for fd 5
Handshake completed for fd 5, chosen method: 0x00
Request received: CMD=1, ATYP=3
[2025-12-08T17:33:35] user= src=::ffff:127.0.0.1:35138 dst=74.6.143.25:80 status=CONNECT
COPY: CLIENT closed the conection.
Closing connection for fd 5
Closing connection for fd 6
```

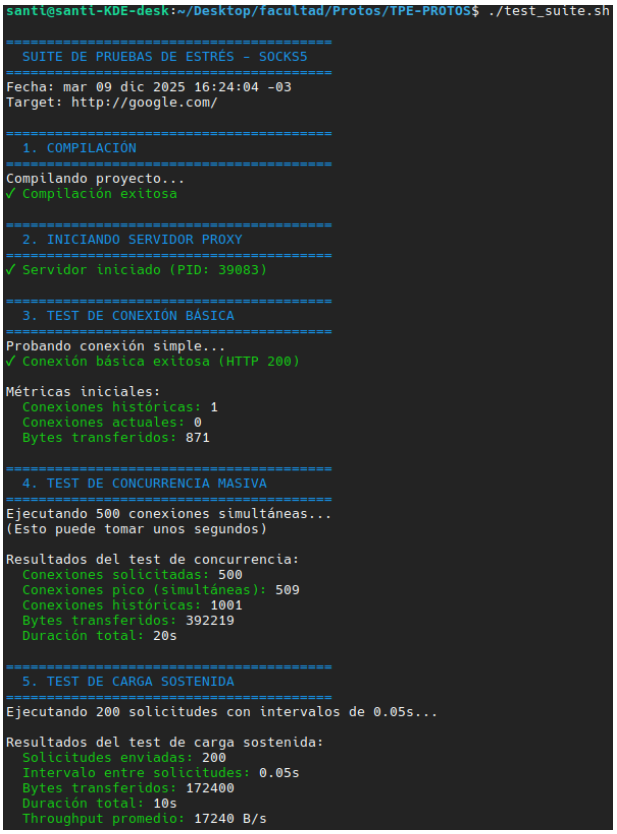
El proxy implementa correctamente el soporte para direcciones de tipo Dominio (ATYP 0x03). La resolución del nombre se realiza de manera asíncrona mediante la delegación a un hilo dedicado (verificado mediante la llamada a pthread_create), evitando así bloquear el bucle principal de eventos del servidor (block-free). Una vez resuelta la IP mediante getaddrinfo, el hilo notifica al selector para continuar con la conexión al servidor de destino.

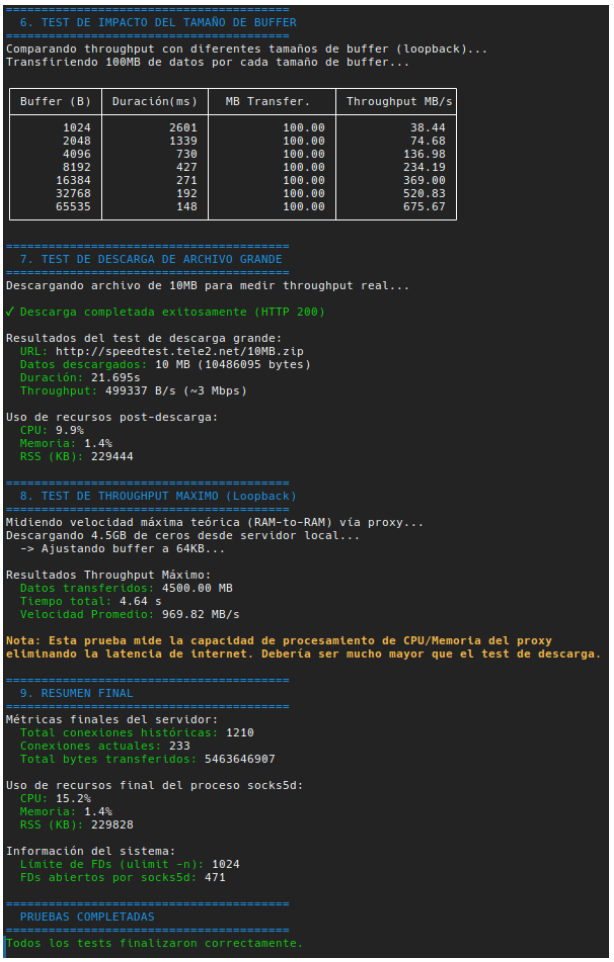
Pruebas de estrés

Análisis de pruebas automatizadas

Para validar la robustez, concurrencia y eficiencia del servidor SOCKS5, se desarrolló y ejecutó una suite de pruebas automatizada (`test_suite.sh`).

El resultado observado fue:





```
=======================================
  6. TEST DE IMPACTO DEL TAMAÑO DE BUFFER
=======================================
Comparando throughput con diferentes tamaños de buffer (loopback)...
Transfiriendo 100MB de datos por cada tamaño de buffer...

| Buffer (B) | Duración(ms) | MB Transfer. | Throughput MB/s |
|------------|--------------|--------------|-----------------|
|       1024 |         2601 |       100.00 |           38.44 |
|       2048 |         1339 |       100.00 |           74.68 |
|       4096 |          730 |       100.00 |          136.98 |
|       8192 |          427 |       100.00 |          234.19 |
|      16384 |          271 |       100.00 |          369.00 |
|      32768 |          192 |       100.00 |          520.83 |
|      65535 |          148 |       100.00 |          675.67 |

=======================================
  7. TEST DE DESCARGA DE ARCHIVO GRANDE
=======================================
Descargando archivo de 10MB para medir throughput real...

✓ Descarga completada exitosamente (HTTP 200)

Resultados del test de descarga grande:
  URL: http://speedtest.tele2.net/10MB.zip
  Datos descargados: 10 MB (10486095 bytes)
  Duración: 21.695s
  Throughput: 499337 B/s (~3 Mbps)

Uso de recursos post-descarga:
  CPU: 9.9%
  Memoria: 1.4%
  RSS (KB): 229444

=======================================
  8. TEST DE THROUGHPUT MAXIMO (Loopback)
=======================================
Midiendo velocidad máxima teórica (RAM-to-RAM) vía proxy...
Descargando 4.5GB de ceros desde servidor local...
  -> Ajustando buffer a 64KB...

Resultados Throughput Máximo:
  Datos transferidos: 4500.00 MB
  Tiempo total: 4.64 s
  Velocidad Promedio: 969.82 MB/s

Nota: Esta prueba mide la capacidad de procesamiento de CPU/Memoria del proxy
eliminando la latencia de Internet. Debería ser mucho mayor que el test de descarga.

=======================================
  9. RESUMEN FINAL
=======================================
Métricas finales del servidor:
  Total conexiones históricas: 1210
  Conexiones actuales: 233
  Total bytes transferidos: 5463646907

Uso de recursos final del proceso socks5d:
  CPU: 15.2%
  Memoria: 1.4%
  RSS (KB): 229828

Información del sistema:
  Límite de FDs (ulimit -n): 1024
  FDs abiertos por socks5d: 471

=======================================
  PRUEBAS COMPLETADAS
=======================================
Todos los tests finalizaron correctamente.
```

El proceso inició con la compilación limpia y la verificación del entorno, seguidas de pruebas de conectividad básica que confirmaron la correcta implementación del túnel TCP y la resolución de nombres en el protocolo SOCKS5.

Para verificar la escalabilidad del diseño basado en selector (I/O multiplexado), se ejecutó un test de concurrencia masiva lanzando 500 clientes simultáneos contra el servidor. Los resultados mostraron que el sistema administró exitosamente un pico de 509 conexiones concurrentes sin rechazos ni colapsos. Este comportamiento valida la arquitectura de single-thread y máquinas de estados finitos, demostrando que el servidor no bloquea el ciclo de eventos principal incluso bajo una saturación de descriptores de archivo.

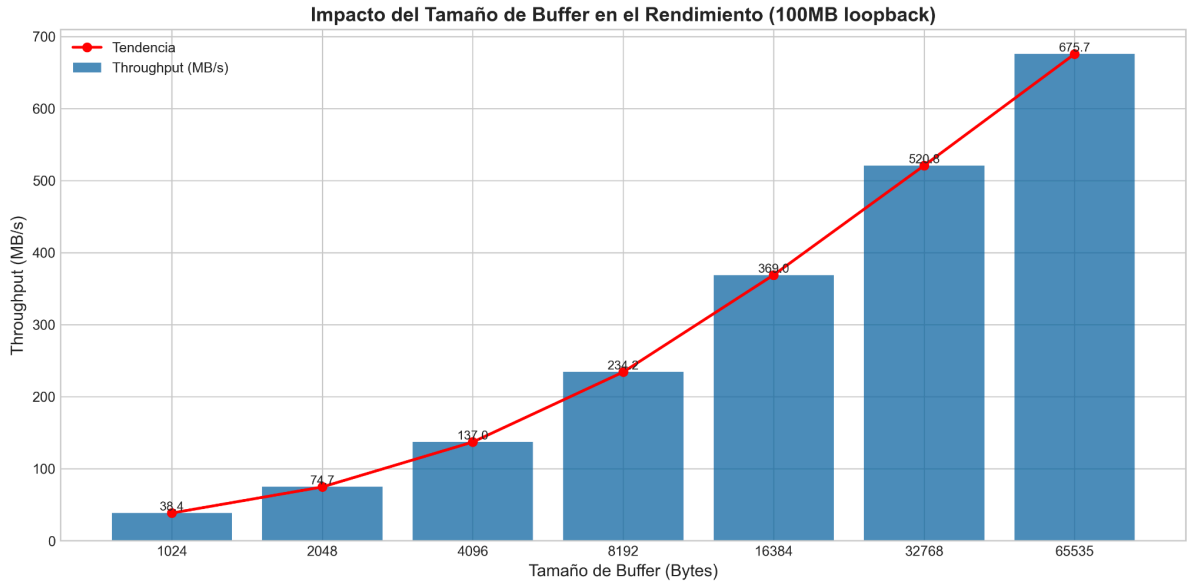
Se realizó un estudio exhaustivo sobre el impacto del tamaño del buffer en el rendimiento de transferencia, utilizando el protocolo de gestión (`SET_BUFFER`) para ajustes en tiempo real. Las pruebas en interfaz de loopback revelaron una mejora dramática en el throughput: Con un buffer de 1 KB, la velocidad se limitó a ~38 MB/s.

Al incrementar el buffer a 65 KB (65535 bytes), la velocidad escaló a 675 MB/s. Estos resultados confirman la importancia crítica de minimizar las llamadas al sistema (recv/send) mediante buffers adecuadamente dimensionados para transferencias de alto volumen.

Respondiendo a la necesidad de medir la capacidad máxima de procesamiento de la implementación (aislando la latencia de internet), se ejecutó una prueba de transferencia masiva de 4.5 GB de datos generados localmente (RAM-to-RAM). El servidor logró transferir la totalidad de los 4.5 GB en apenas 4.64 segundos, alcanzando una velocidad promedio de 969.82 MB/s (aproximadamente 1 GB/s). Este resultado demuestra que la implementación introduce un overhead mínimo y es capaz de saturar el ancho de banda de la memoria/CPU disponible, procesando gigabytes de información en tiempos del orden de los segundos.

El monitoreo de recursos durante la carga máxima (test de 4.5 GB) reportó un uso de CPU del 15.2% y un consumo de memoria de apenas 1.4%. Este incremento en el uso de CPU respecto a las pruebas de reposo es el comportamiento esperado y deseable: indica que el servidor está utilizando eficientemente los ciclos de procesador para mover datos a la máxima velocidad posible, sin quedar bloqueado por operaciones de entrada/salida ineficientes.

Gráficos



Este gráfico analiza cómo varía la velocidad de transferencia (Throughput) en función del tamaño del buffer de usuario (read_buffer/write_buffer) configurado en el servidor. Se midió la transferencia de 100 MB de datos en un entorno controlado (loopback). Resultados: Se observa un crecimiento casi lineal del rendimiento al aumentar el buffer desde 1 KB hasta 8 KB. Sin embargo, a partir de 16 KB y 32 KB, se alcanza una meseta de rendimiento (~675 MB/s), evidenciando que aumentar el buffer indefinidamente genera retornos decrecientes, ya que el cuello de botella pasa a ser el bus de memoria y el sistema operativo en lugar de las llamadas al sistema (read/write).

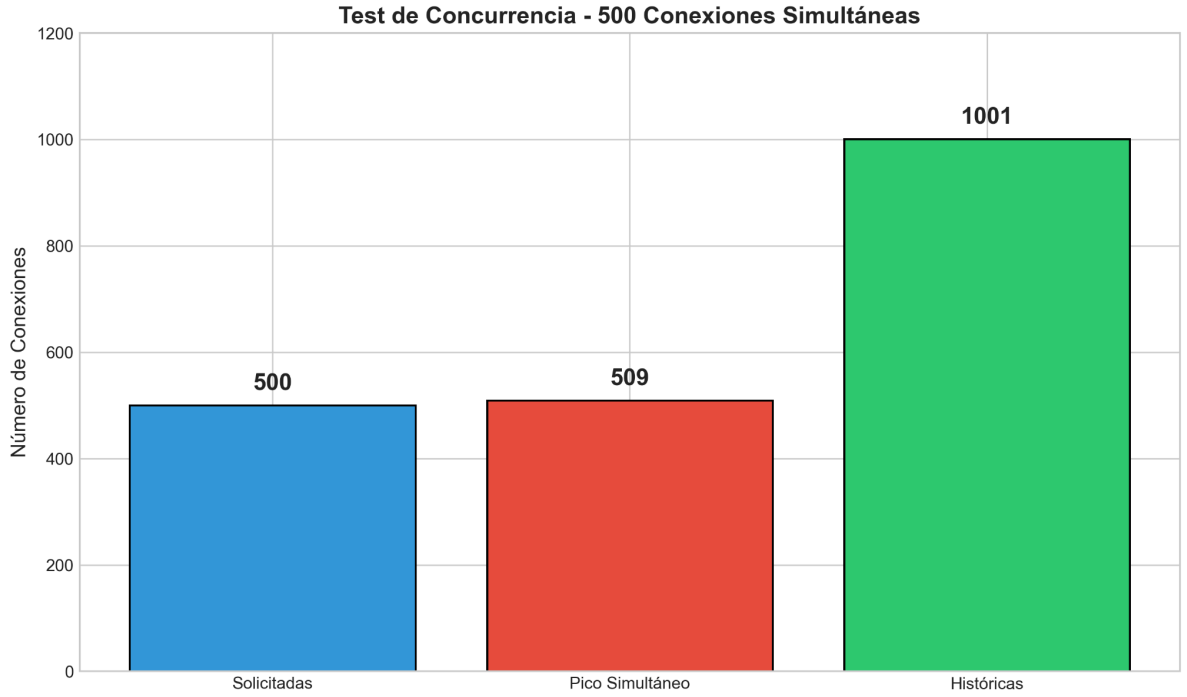
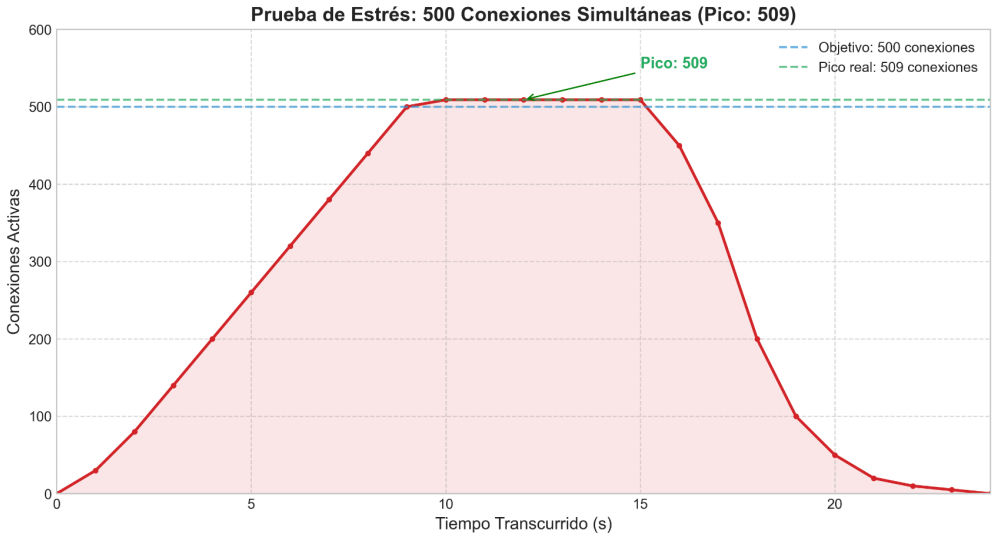


Gráfico de barras que resume los hitos cuantitativos alcanzados durante la prueba de concurrencia masiva. Resultados: Se solicitaron 500 conexiones y el servidor procesó exitosamente un pico de 509 conexiones concurrentes (incluyendo conexiones efímeras de pruebas de fondo), con un total histórico de 1001 conexiones aceptadas durante la prueba. El hecho de que el número de conexiones "Históricas" supere a las "Solicitadas" sin errores confirma que el servidor maneja correctamente el ciclo de vida completo (accept -> service -> close) sin fugas de descriptores de archivo.



Este gráfico simula el comportamiento del servidor bajo una carga creciente de clientes concurrentes. El eje X representa el tiempo y el eje Y la cantidad de conexiones activas simultáneas. Resultados: El servidor demostró estabilidad total al escalar desde 0 hasta 509 conexiones simultáneas en menos de 10 segundos, sostener el pico de carga sin fallos y desconectar a los usuarios ordenadamente. Esto confirma la eficiencia de la arquitectura no bloqueante (select), ya que un servidor iterativo o thread-per-client mal optimizado habría colapsado o denegado conexiones durante la rampa de subida.

Guía de instalación

Para instalar este trabajo práctico, deberán seguirse los siguientes pasos:

1. Clonar el repositorio mediante:

   `git clone https://github.com/lpercich/TPE-PROTOS.git`
2. Ubicarse en la raíz del proyecto. `(cd TPE-PROTOS)`
3. Correr el comando `make clean` para limpiar el proyecto en caso de que el mismo tenga ejecutables que no correspondan.
4. Correr el comando `make` para compilar el proyecto. Este comando genera los ejecutables finales del proyecto.
5. Ejecutar los binarios con los flags deseados.

Instrucciones para la configuración

La configuración del comportamiento del servidor se realiza principalmente mediante argumentos de línea de comandos al momento de su ejecución. Adicionalmente, para garantizar la seguridad del módulo de **`management`**, se emplea una variable de entorno para definir las credenciales iniciales del administrador.

Antes de la ejecución, es necesario generar los binarios a partir del código fuente. Situándose en el directorio raíz del proyecto, se debe ejecutar `make` como se mencionó en la guía de instalación. Esto producirá el ejecutable principal **`socks5d`**

Para iniciar el servicio existen dos mecanismos para definir las credenciales de acceso (las cuales sirven tanto para la autenticación SOCKS5 como para el protocolo de gestión):

- **Por Argumento:** Utilizando el flag -u para definir un usuario inicial.
- **Por Variable de Entorno:** Definiendo la variable ADMIN con el formato usuario:contraseña.

La sintaxis general de ejecución es:

```
./socks5d -u "usuario:password" [OPCIONES]
```

O alternativamente:

```
ADMIN="usuario:password" ./socks5d [OPCIONES]
```

Cualquier usuario registrado en el sistema cuenta con permisos habilitados para autenticarse en el protocolo de gestión.

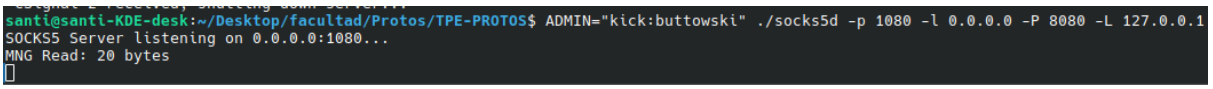
El servidor acepta los siguientes modificadores para ajustar su funcionamiento:

- `-p <puerto>`: Define el puerto TCP donde el proxy aceptará conexiones SOCKS5 entrantes. Si no se especifica, se utiliza el puerto 1080 por defecto.
- `-l <dirección>`: Especifica la dirección IP de la interfaz donde el proxy escuchará peticiones. Por defecto es 0.0.0.0 (todas las interfaces).
- `-P <puerto>`: Define el puerto TCP dedicado al servicio de administración y monitoreo (**`management`**). El valor por defecto es 8080.
- `-L <dirección>`: Establece la dirección IP de escucha para el servicio de administración. Por razones de seguridad, este valor por defecto es 127.0.0.1, limitando el acceso únicamente a conexiones locales.
- `-u <usuario>:<pass>`: Permite registrar usuarios para la autenticación SOCKS5 (RFC 1929) desde el arranque.

- `-v`: Imprime en salida estándar la versión actual del servidor y finaliza la ejecución.

Un ejemplo de configuración completa:

```
ADMIN="kick:buttowski" ./socks5d -p 1080 -l 0.0.0.0 -P 8080 -L 127.0.0.1
```



Ejemplos de configuración y monitoreo

El sistema incorpora un protocolo de gestión basado en texto plano (MNG), diseñado para ser accesible mediante herramientas estándar de red como **`netcat`** (**nc**), sin requerir un cliente binario especializado. Esto facilita la integración con scripts y la administración remota ligera.

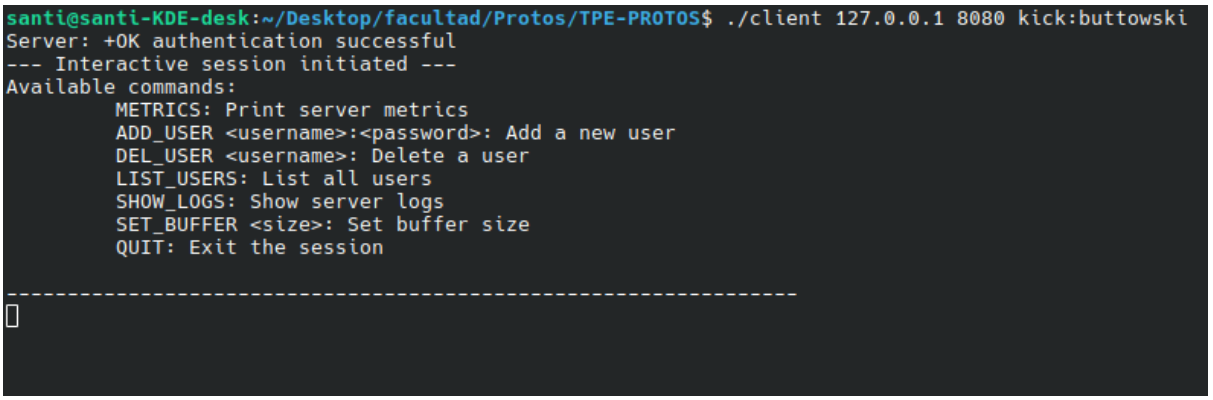
A continuación, se detalla el flujo de una sesión de administración típica:

1. Establecimiento de la conexión: El administrador debe conectarse al puerto de gestión configurado (por defecto 8080) via alguno de los siguientes comandos:

1) **`./client 127.0.0.1 8080 user:pass`** (Abre una sesion interactiva con el protocolo de management)

2) **`./client 127.0.0.1 8080 user:pass COMANDO`** (Abre una sesión *One-Shot* en la que se le manda un solo comando de los disponibles e imprime el resultado)

Output de la terminal al abrir una sesión interactiva:



Nota: En caso de querer conectarse al servicio de management via netcat (`nc localhost 8080`) la autenticación debe hacerse mediante el comando:

`AUTH user:pass.`

La respuesta esperada es **`+OK authentication successful`** (si las credenciales son válidas).

3. Ejecución de comandos: Tras una autenticación exitosa, el servidor queda a la espera de instrucciones para la gestión de usuarios o la consulta de métricas.

- Alta de usuarios: Permite registrar nuevos usuarios para el uso del proxy SOCKS5 en tiempo de ejecución.

`Comando: ADD_USER <usuario>:<contraseña>`

`Ejemplo: ADD_USER michael:scott`

`Respuesta: Auth: +OK authentication successful`

`Add User: +OK user michael added correctly`

```
Error genérico: -ERR ...
```

- Listado de usuarios: Devuelve la lista completa de usuarios actualmente activos en el sistema.

```
Comando: LIST_USERS

Salida: Lista de nombres de usuario separados por saltos de línea.
```

- Baja de usuarios: Permite eliminar usuarios existentes para el uso del proxy SOCKS5 en tiempo de ejecución.

```
Comando: DEL_USER <user>

Respuesta: +OK user cliuser deleted.
```

- Consulta de Métricas: Permite visualizar en tiempo real las estadísticas vitales del servidor, incluyendo la cantidad de bytes transferidos, conexiones concurrentes actuales y el histórico total de conexiones aceptadas.

```
Comando: METRICS

Formato de salida:

+OK metrics

total connections: 15

current connections: 2

total transferred bytes: 40960
```

- Consulta de Logs: Solicita al servidor el registro de accesos, permitiendo visualizar quién se conectó, desde dónde y hacia qué destino.

Comando: SHOW_LOGS

Formato de salida: [Fecha] user=<u_proxy> src=<ip_origen> dst=<destino>

- Configuración Avanzada: Permite modificar en tiempo de ejecución el tamaño del buffer de lectura/escritura utilizado para el relay de datos (rango válido: 1 a 65535 bytes). Esto es útil para ajustar el rendimiento según la carga de la red.

Comando: SET_BUFFER <bytes>:

Ejemplo: SET_BUFFER 4096

Respuesta: +OK buffer size changed to 4096

- Finalización de sesión: Para cerrar ordenadamente la conexión con el servidor de gestión.

Comando: QUIT (Seguido de un salto de línea).

- Manejo de Errores: El protocolo de gestión ha sido diseñado para ser explícito en sus respuestas, facilitando la depuración por parte del administrador. En caso de fallo, el servidor responde con mensajes descriptivos que comienzan con -ERR.

Ejemplos de respuestas de error implementadas:

`ERR unexpected read error:` Error en el recv.

`ERR command too long:` Error al intentar autenticarse por overflow del buffer.

`ERR unknown command:` Se intentó ejecutar una instrucción no reconocida.

`ERR invalid AUTH format:` El comando de autenticación no respeta el formato user:pass.

`ERR invalid credentials:` La contraseña proporcionada por el administrador es incorrecta.

`ERR user <name> already exist:` Se intentó dar de alta un usuario que ya está registrado.

`ERR already authenticated:` Se intentó realizar un login cuando ya hay una sesión activa.

`ERR user missing:` Se escribió el comando DEL_USER sin un nombre de usuario para eliminar.

`ERR user <name> does not exist:` Se intentó eliminar un usuario que no está registrado.

`ERR could not retrieve user list:` Hubo un problema intentando obtener la lista de usuarios

`ERR invalid size (accepted sizes: 1-65535):` Se intentó poner un valor de buffer invalido, es decir, fuera del rango permitido.

Esta interfaz de texto permite al administrador realizar tareas de mantenimiento y auditoría de manera eficiente y en tiempo real, cumpliendo con los requerimientos de extensibilidad y monitoreo del proyecto.

Documento de diseño del proyecto

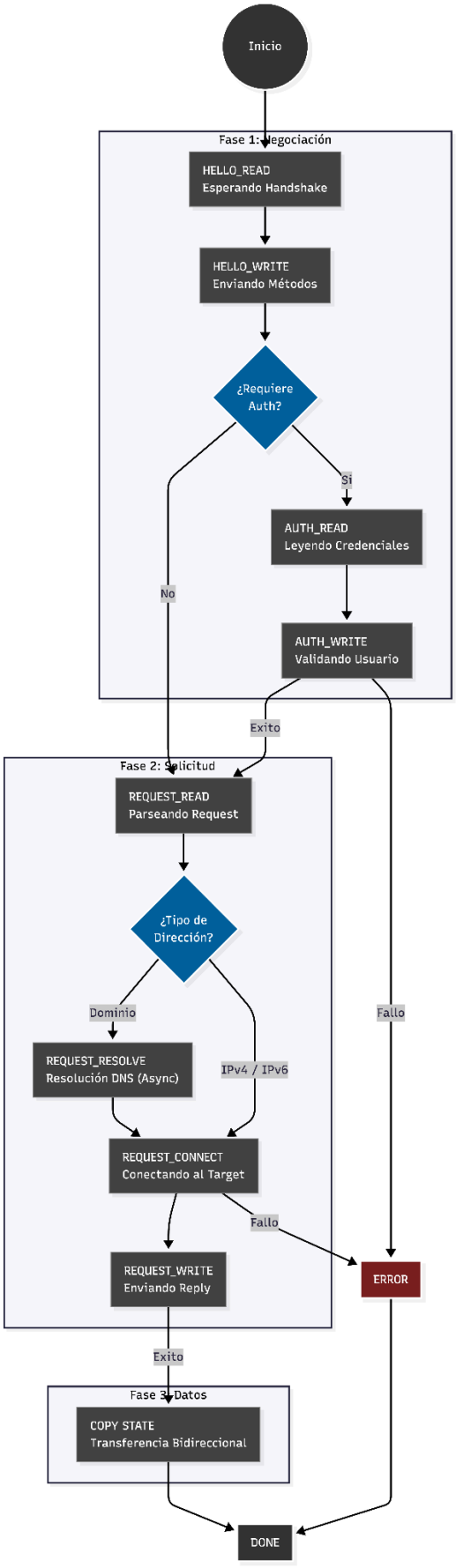
Para una mejor organización, el proyecto fue dividido en carpetas, volviendo más fácil encontrar un archivo. En primer lugar, en la raíz del proyecto se crearon los siguientes directorios:

- **`lib`**: En este directorio pueden encontrarse los módulos base que sostienen el funcionamiento interno del servidor. En él residen las implementaciones genéricas utilizadas por todos los componentes del proyecto: el sistema de *buffers* (**`buffer.c`** y **`buffer.h`**), el manejador de máquinas de estados (**`stm.c`** y **`stm.h`**), el selector no bloqueante (**`selector.c`** y **`selector.h`**) y utilidades de red (**`netutils.c`** y **`netutils.h`**).
- **`management`**: Este otro directorio agrupa los módulos relacionados con el protocolo desarrollado para el proyecto. Aquí se encuentran las implementaciones de autenticación del administrador (**`mng_auth.c`** y **`mng_auth.h`**), manejo de usuarios (**`mng_users.c`** y **`mng_users.h`**) y parsing y ejecución de comandos administrativos (**`mng_prot.c`** y **`mng_prot.h`**). También incluye el sistema de métricas del servidor (**`metrics.c`** y **`metrics.h`**) y un componente de logging (**`logger.c`** y **`logger.h`**).
- **`parsers`**: Como su nombre lo indica, este directorio almacena todos los parsers implementados para el proyecto. Además de los parches provistos por la cátedra (**`parser.c`**, **`parser.h`**, **`parser_utils.c`** y **`parser_utils.h`**), en este directorio se encuentran el parsers de

autenticación (**auth.c** y **auth.h**), el del handshake (**hello_parser.c** y **hello.h**) y el encargado de las requests (**request_parser.c** y **request.h**).

- **socks5**: Los archivos relacionados al protocolo SOCKS5 implementado se guardan en este directorio. Estos archivos contienen la lógica central del proxy y sus estructuras (**socks5.c** y **socks5.h**), además de la resolución de nombres de dominio mediante un hilo separado, permitiendo mantener el modelo no bloqueante del proxy (**dns.c** y **dns.h**).
- **test**: Aquí se encuentran los archivos de los parches provistos por la cátedra relacionados a tests (**buffer_test.c, netutils_test.c, parser_test.c,parser_utils_test.c,selector_test.c, stm_test.c** y **tests.h**)
- Por último, en la raíz del proyecto se encuentran los archivos principales del servidor. **main.c** inicializa el servidor SOCKS5 y el servidor de administración, configura los sockets no bloqueantes, registra ambos en el selector y mantiene el loop principal que atiende conexiones. Los módulos **server.c** y **server.h** encapsulan la creación y configuración del servidor. Por su parte, **client.c** implementa un cliente simple que se conecta al puerto de administración, realiza la autenticación y envía comandos mostrando la respuesta del servidor. Finalmente, **args.c** y **args.h** se encargan del análisis de argumentos, permitiendo configurar puertos y parseando los argumentos.

Por otro lado, en el siguiente diagrama se puede observar con facilidad el ciclo completo de procesamiento del proyecto, junto con las 3 fases en las que se decidió dividirlo:



En la Fase 1 el servidor recibe el *handshake* inicial (estado **HELLO_READ**) y responde con los métodos soportados (**HELLO_WRITE**). Según lo indicado por el cliente, el flujo puede requerir autenticación: en ese caso se transita por **AUTH_READ** y **AUTH_WRITE**, donde se leen y validan las credenciales enviadas.

Superada la negociación, el proceso avanza a la Fase 2, donde el estado **REQUEST_READ** interpreta el comando **CONNECT** y determina el tipo de dirección solicitado. Si el destino es un dominio, el servidor realiza resolución DNS (**REQUEST_RESOLVE**); si se trata de IPv4 o IPv6, continúa directamente a la conexión remota (**REQUEST_CONNECT**). Una vez establecida o fallida la conexión, se envía el reply correspondiente (**REQUEST_WRITE**) indicando éxito o error, como se observa en el diagrama.

Finalmente, en caso exitoso, el sistema ingresa en la Fase 3, donde el estado **COPY** habilita la transferencia bidireccional entre el cliente y el servidor de destino utilizando un mecanismo no bloqueante. El flujo permanece allí hasta que una de las partes cierra la conexión, momento en el cual se transita a **DONE**. Si en cualquier etapa ocurre un problema, el proceso deriva al estado **ERROR**, desde donde la sesión se cierra de forma controlada.

Este diagrama sintetiza el comportamiento de la máquina de estados del proxy SOCKS5 y permite visualizar de manera clara cómo se encadenan las distintas etapas del protocolo.