Implementación de un Servidor Proxy SOCKS5 No Bloqueante

Grupo 13
Materia: Protocolos de Comunicación

12 de febrero de 2026

Índice

1. Descripción detallada de los protocolos y aplicaciones desarrolladas

En este trabajo práctico se implementó un servidor proxy basado en el protocolo SOCKS versión 5, según las especificaciones de los RFCs 1928 y 1929. El servidor permite que múltiples clientes establezcan conexiones TCP a través del proxy de manera transparente, manejando autenticación opcional, resolución de nombres de dominio y tunelización completa de datos entre cliente y origen. Toda la arquitectura está desarrollada bajo un modelo completamente no bloqueante, utilizando un único loop de eventos basado en `select` mediante la librería `selector` provista por la cátedra.

1.1. Protocolos implementados

SOCKS5 (RFC 1928)

El servidor soporta el flujo estándar de SOCKS5:

- Negociación de métodos (HELLO / METHOD SELECTION).
- Autenticación opcional según RFC 1929.
- Procesamiento del mensaje REQUEST (comando CONNECT).
- Envío del mensaje REP con el código de resultado correspondiente.
- Tunelización TCP transparente entre el cliente y el servidor destino.

Se implementa soporte para las variantes de dirección requeridas:

- IPv4 (ATYP = 0x01).
- Nombres de dominio (ATYP = 0x03).
- IPv6 (ATYP = 0x04).

Autenticación (RFC 1929)

El proxy permite autenticar usuarios mediante el método `USER/PASS`. Las credenciales se especifican por línea de comandos utilizando el parámetro `-u usuario:clave`. El servidor valida estas credenciales antes de permitir el envío del mensaje REQUEST. En caso de error, se informa mediante un código de autenticación fallida y se cierra la conexión.

Resolución de nombres de dominio

Para direcciones del tipo FQDN (ATYP = 0x03), la resolución de nombres se realiza mediante un módulo de resolución DNS asíncrono. Este módulo implementa una cola de

trabajos protegida por mutex y un conjunto de hilos de resolución (entre 2 y 4, configurables). Cada consulta se encola sin bloquear y es procesada por alguno de los hilos.

Cuando un hilo obtiene una respuesta, notifica al `selector` a través de un descriptor de notificación no bloqueante. El callback asociado recupera el resultado y continúa la FSM del REQUEST sin detener el loop principal. Si el FQDN resuelve a múltiples direcciones IP, se intenta conectar cada una en orden hasta encontrar una válida.

Este diseño permite que el servidor continúe aceptando clientes y manejando túneles mientras una o varias resoluciones DNS están pendientes, manteniendo el comportamiento completamente no bloqueante requerido.

1.2. Arquitectura de la aplicación

Máquinas de estados

Cada conexión cliente se modela mediante dos máquinas de estados independientes:

- **STM del cliente**: maneja HELLO, autenticación, REQUEST, envío de REP y la mitad cliente del túnel.
- **STM del origen**: maneja el establecimiento de la conexión no bloqueante con el servidor destino y la mitad origin del túnel.

La combinación de ambas asegura que todas las operaciones sean no bloqueantes y que la transición entre etapas del protocolo sea clara y modular.

Tunelización

Una vez establecida la conexión con el destino, el servidor habilita un modo de túnel full–duplex. Se utilizan dos buffers independientes para transferir datos:

- Cliente → Origen.
- Origen → Cliente.

El servidor transmite datos en cuanto los sockets están listos para lectura/escritura, sin bloquear y sin interpretar el contenido.

Monitor y métricas

El proyecto incluye un segundo servicio TCP para monitoreo, también no bloqueante, que expone estadísticas operativas:

- Conexiones totales y concurrentes.
- Bytes transferidos en cada sentido.
- Códigos REP enviados.

- Autenticaciones exitosas y fallidas.
- Resoluciones DNS exitosas y fallidas.

La consulta del monitor devuelve un snapshot legible que permite evaluar el comportamiento del proxy en tiempo real.

Registro de accesos

Tal como pide el enunciado, el proxy mantiene un registro de accesos donde se almacena:

- Usuario autenticado (o "unauthenticated").
- Dirección de origen.
- Dirección y puerto del destino.
- Timestamp.
- Resultado del intento de conexión.

Este registro permite auditar actividades en caso de incidentes o reclamos.

2. Problemas encontrados durante el diseño y la implementación

A lo largo del desarrollo del proxy SOCKS5 surgieron varios desafíos técnicos que requirieron replanteos y ajustes incrementales sobre la arquitectura inicial.

Gestión correcta de descriptores y cierre de conexiones

Las primeras pruebas de stress mostraban un crecimiento sostenido de descriptores abiertos, lo que evidenciaba fugas de recursos. El origen del problema era la combinación de cierres parciales (mediante `shutdown`) y estados terminales que no siempre liberaban la estructura de conexión. Reescribir de forma clara el ciclo de vida de una conexión evitó fugas y permitió estabilizar las pruebas de carga.

Integración de parámetros de configuración

La migración al sistema de argumentos provisto por la cátedra requirió revisar varias partes del servidor que tenían valores fijos (direcciones, puertos, usuarios). Algunos módulos dependían de rutas o configuraciones internas y debieron adaptarse para usar exclusivamente `args`. Este cambio permitió cumplir con el requisito de que el servidor sea configurable sin recompilar. La principal dificultad vino de no integrar args desde el inicio del desarrollo del proyecto.

3. Limitaciones de la aplicación

Si bien el proxy cumple con los requisitos funcionales del trabajo práctico y opera de manera estable bajo carga, presenta algunas limitaciones derivadas del alcance del diseño y de las decisiones tomadas durante la implementación.

Sin inspección avanzada del tráfico

El proxy tuneliza datos de manera transparente y no implementa dissectores de protocolos de aplicación (HTTP, POP3, etc.). El contenido del tráfico no se interpreta y no se generan estadísticas específicas por tipo de protocolo.

Resolución DNS dependiente del sistema

La resolución de FQDN se realiza mediante `getaddrinfo` dentro de un conjunto de hilos dedicados. Esto implica que el comportamiento frente a timeouts o fallos depende de la implementación del sistema operativo. Si bien se evita el bloqueo del servidor, no existe un control fino sobre los tiempos de resolución ni un mecanismo de cancelación anticipada.

4. Posibles extensiones

Existen varias mejoras que podrían incorporarse para ampliar las capacidades del proxy y facilitar su uso en escenarios más complejos. Algunas de ellas requieren cambios modestos sobre la arquitectura actual y otras implicarían rediseños más profundos.

Rotación y persistencia mejorada de logs

El sistema de registro podría ampliarse con:

- rotación automática del archivo de accesos,
- configuración del nivel de detalle,
- exportación en formatos estructurados (JSON o CSV),
- timestamps con mayor precisión o sincronización.

Mayor robustez y escalabilidad

Si bien el diseño basado en `select` es adecuado para el volumen requerido por el trabajo práctico, podría migrarse a modelos más eficientes. También serían posibles extensiones como balanceo entre múltiples hilos de trabajo o la capacidad de ejecutar varios procesos coordinados.

Estas extensiones no forman parte del alcance del trabajo práctico, pero surgen de manera natural del diseño actual y permitirían evolucionar el proxy hacia un entorno más flexible y cercano a un servidor de producción.

5. Conclusiones

La implementación del servidor SOCKS5 permitió integrar en una misma aplicación varios conceptos centrales de la materia: comunicación no bloqueante, resolución asíncrona de nombres, manejo seguro de concurrencia y construcción de protocolos sobre TCP. El desarrollo evidenció la importancia de estructurar claramente el ciclo de vida de una conexión y de evitar bloqueos en todas las etapas, especialmente bajo carga elevada.

El uso de un único loop de eventos demostró ser adecuado para el volumen de conexiones esperado, y el diseño resultó lo suficientemente flexible como para incorporar módulos adicionales, como el monitor y el sistema de registro de accesos. A partir de esta arquitectura fue posible agregar funcionalidades sin comprometer la estabilidad del proxy ni su comportamiento no bloqueante.

En conjunto, el proyecto permitió comprender en profundidad el funcionamiento interno de un proxy de propósito general y los desafíos prácticos asociados a su implementación.

6. Ejemplos de prueba

En esta sección se describen las pruebas funcionales y de carga utilizadas para validar el correcto funcionamiento del servidor SOCKS5. Todas las pruebas se ejecutaron en un entorno local, utilizando direcciones IPv4 y sin autenticación habilitada (lo que es consistente con los valores observados en las métricas).

Pruebas funcionales básicas

Se verificó el funcionamiento del proxy con clientes estándar como `curl` utilizando la opción `--socks5`. Entre las pruebas realizadas se incluyeron:

- Handshake HELLO y selección correcta del método.
- Envío del REQUEST `CONNECT` a servidores HTTP locales.
- Confirmación de que el túnel opera en forma full–duplex.
- Soporte de direcciones IPv4, IPv6 y FQDN (utilizando el resolvedor DNS asíncrono del proyecto).

Todas las pruebas funcionales básicas resultaron satisfactorias.

Pruebas de concurrencia y stress

Se utilizó un generador de carga basado en Python que lanza múltiples *workers*, cada uno haciendo varias solicitudes consecutivas a través del proxy. Para cada solicitud, el cliente realiza:

1. Handshake SOCKS5 (HELLO)
2. REQUEST con destino IPv4
3. Tunelización de un HTTP GET

Este patrón simula conexiones reales de baja duración pero alta frecuencia.

El servidor de pruebas fue sometido a dos escenarios principales: 500 y 1000 *workers* ejecutando 10 solicitudes cada uno. Las métricas fueron obtenidas a través del monitor del proxy.

Escenario 1: 500 workers (5000 conexiones)

- Tiempo total: 11.71 s
- Conexiones exitosas: 5000 / 5000 (100 %)
- Latencias: min 259 ms, media 599 ms, p95 942 ms
- Métricas reportadas:

```
max_concurrent_connections: 517
total_connections:          6000
rep[0x00]:                  6000
```

Escenario 2: 1000 workers (10000 conexiones)

- Tiempo total: 29.36 s
- Conexiones exitosas: 0 / 10000 (0 %)
- Errores reportados por el cliente: códigos HTTP 0 y 429
- Métricas reportadas:

```
max_concurrent_connections: 852
total_connections:          15710
rep[0x00]:                  14684
rep[0x01]:                   760
```

En este escenario se observa una clara saturación del servidor. Si bien logra aceptar y procesar parcialmente un volumen superior al requerido (500 concurrentes), el valor de `max_concurrent_connections` se estabiliza alrededor de 850.

Degradación del throughput

Comparando ambos escenarios:

- Con 500 concurrentes, el sistema responde con throughput estable.
- Con 1000 concurrentes, el tiempo total casi se triplica, y la tasa de éxito cae drásticamente.

El comportamiento observado es consistente con la arquitectura basada en un único *event loop*: a medida que el número de conexiones simultáneas se acerca al límite del sistema, aumentan las colisiones en el selector y la competencia por recursos.

Máxima cantidad de conexiones simultáneas soportadas

Basado en los resultados del monitor:

- **500 conexiones simultáneas**: manejadas sin problemas.
- **Máximo registrado**: **852** conexiones concurrentes estables.

Conclusión de las pruebas

Las pruebas funcionales y de carga confirman que el proxy:

- Maneja correctamente múltiples clientes concurrentes.
- Mantiene throughput estable hasta aproximadamente 500 conexiones activas.
- Soporta más de 800 conexiones simultáneas antes de degradarse.
- Reporta métricas consistentes que permiten analizar su comportamiento interno.

7. Guía de instalación

Esta sección describe los pasos necesarios para compilar y ejecutar el servidor SOCKS5. No es necesario utilizar herramientas externas ni instalar dependencias adicionales más allá de un compilador estándar de C y `make`. El proyecto incluye un `Makefile` que automatiza todo el proceso de construcción.

Requerimientos

- Compilador `gcc` compatible con C11.
- Sistema operativo tipo UNIX.
- Herramientas `make` y librería de threads `pthread`.

Estructura del proyecto

El código fuente se encuentra en el directorio `src/`. El `Makefile` genera automáticamente dos carpetas:

- `build/`: contiene los archivos objeto intermedios.
- `bin/`: contiene el ejecutable final `echo_server`.

Compilación

Para compilar el servidor, basta con ejecutar:

```
$ make
```

El comando compila todos los archivos fuente utilizando los flags:

- `-Wall -Wextra -pedantic`: advertencias estrictas.
- `-std=c11`: estándar C11.
- `-pthread`: soporte para hilos en el módulo de resolución DNS.
- `-g`: inclusión de símbolos de depuración.

Si la compilación es exitosa, se genera el binario:

```
bin/echo_server
```

Parámetros de línea de comandos

El servidor utiliza los argumentos definidos en el módulo `args` para configurar la dirección y puerto del proxy SOCKS5, el servicio de management y los usuarios válidos. La función `parse_args()` inicializa la estructura `socks5args` con los siguientes valores por defecto: `socks_addr = "0.0.0.0"`, `socks_port = 1080`, `mng_addr = "127.0.0.1"` y `mng_port = 8080`.:contentReference[oaicite:2]index=2

Los parámetros disponibles son:

- `-h`: imprime un mensaje de ayuda con el *usage* y termina la ejecución.
- `-l <SOCKS addr>`: dirección donde servirá el proxy SOCKS (por ejemplo, `0.0.0.0` o `127.0.0.1`). Por defecto `0.0.0.0`.
- `-p <SOCKS port>`: puerto entrante para conexiones SOCKS. Debe estar en el rango 1–65535; el valor por defecto es `1080`.
- `-L <conf addr>`: dirección donde servirá el servicio de management (monitor). Por defecto `127.0.0.1`.

- `-P <conf port>`: puerto entrante para conexiones del servicio de monitor; el valor por defecto es `8080`.
- `-u <name>:<pass>`: usuario y contraseña habilitados para usar el proxy. Se pueden especificar hasta `MAX_USERS` entradas (10 usuarios como máximo).
- `-v`: imprime información de versión del servidor y termina sin iniciar el proxy.

Un ejemplo de ejecución típica es:

```
$ ./bin/echo_server -l 0.0.0.0 -p 1080 \
    -L 127.0.0.1 -P 8080 \
    -u user1:pass1 -u admin:admin
```

En este ejemplo, el proxy SOCKS5 escucha en `0.0.0.0:1080`, el monitor en `127.0.0.1:8080` y se configuran dos usuarios válidos para autenticación.

Ejecución

Luego de compilar, el servidor puede ejecutarse directamente con:

```
$ ./bin/echo_server [opciones]
```

o utilizando la regla conveniente del Makefile:

```
$ make run ARGS="[opciones]"
```

En todos los casos, los parámetros de configuración deben pasarse por línea de comandos según lo explicado en la sección anterior.

Limpieza de archivos compilados

Para eliminar los archivos objeto y los binarios generados:

```
$ make clean
```

Este comando elimina las carpetas `build/` y `bin/`, dejando el proyecto en estado limpio.

8. Instrucciones para la configuración

El servidor SOCKS5 se configura exclusivamente mediante parámetros de línea de comandos y, en tiempo de ejecución, a través del monitor. La configuración inicial se define al momento de ejecución, mientras que ciertos ajustes pueden realizarse dinámicamente sin reiniciar el servidor.

Parámetros de línea de comandos

Los argumentos disponibles son:

- `-l <addr>`: dirección donde escucha el servidor SOCKS5 (por defecto `0.0.0.0`).
- `-p <port>`: puerto del servicio SOCKS5 (por defecto 1080).
- `-L <addr>`: dirección del servicio de monitoreo.
- `-P <port>`: puerto del servicio de monitoreo (por defecto 8080).
- `-u <user:pass>`: agrega un usuario válido para autenticación. Puede especificarse varias veces.
- `-h`: muestra ayuda y finaliza.
- `-v`: muestra la versión del servidor.

Configuración en tiempo de ejecución

El monitor permite modificar ciertos aspectos del servidor sin reiniciar la aplicación. Actualmente se encuentran disponibles:

- `RESET`: reinicia todas las métricas acumuladas.
- `ADDUSER <user><pass>`: agrega un nuevo usuario válido para autenticación.

Ambos comandos se envían conectándose al puerto de monitoreo mediante `nc` o herramientas similares.

9. Ejemplos de configuración y monitoreo

Un ejemplo de configuración típica es:

```
./bin/echo_server -l 0.0.0.0 -p 1080 \
    -L 127.0.0.1 -P 8080 \
    -u admin:admin -u user1:1234
```

Un ejemplo de reseteo de métricas en runtime es:

```
echo "RESET" | nc 127.0.0.1 8080
```

Y por último para agregar un usuario basta con correr:

```
echo "ADDUSER user pass123" | nc 127.0.0.1 8080
```

10. Documento de diseño del proyecto

Antes de empezar a programar el proxy, nos tomamos un tiempo para pensar cómo queríamos organizar el servidor. La idea general fue evitar que el código se volviera un

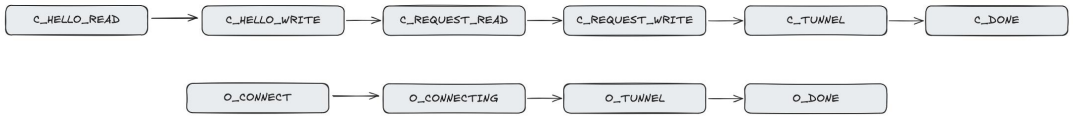

Figura 1: STM's

gran "if/else" caótico y, al mismo tiempo, mantenerlo lo suficientemente simple como para poder depurarlo y extenderlo sin problemas.

El corazón del servidor es un único *event loop* basado en `select`. A partir de eso decidimos que cada conexión SOCKS5 debía tener sus propias máquinas de estados: una para lo que pasa del lado del cliente y otra para lo que pasa del lado del servidor de destino. Esto nos permitió separar las etapas del protocolo y nos facilitó mucho detectar errores de transición mientras implementábamos.

A nivel conceptual, el diseño de las dos máquinas quedó más o menos así:

Con estas dos máquinas pudimos estructurar todo el flujo: desde el HELLO hasta el túnel final. También nos ayudó a que cada transición fuera explícita: cada estado hace una sola cosa y después cede el control al `select`. Esto hizo que el servidor se mantuviera no bloqueante sin necesidad de usar hilos en la parte principal.

La tunelización la pensamos como una transferencia de datos "transparente". Usamos dos buffers: uno para los datos que van del cliente al servidor de destino y otro para los que vuelven. El módulo del túnel simplemente mueve bytes cuando el selector indica que puede leer o escribir.

Para la resolución de FQDN decidimos implementar un resolver con hilos. La idea fue simple: un pool de 2 a 4 hilos que corren `getaddrinfo()`, una cola protegida por mutex, y un descriptor de notificación para avisarle al selector cuando una resolución está lista. Esto nos permitió continuar con la ejecución normal del proxy mientras los hilos resolvían dominios en paralelo.

Finalmente, agregamos dos componentes auxiliares: el **monitor**, que expone métricas internas en un socket aparte, y el **registro de accesos**, que guarda la información mínima necesaria para saber quién se conectó a qué destino.

En conjunto, la arquitectura se mantuvo modular y clara. Cada parte cumple un rol acotado y la comunicación entre módulos se da siempre a través del selector o de notificaciones explícitas. El resultado final es un proxy que podemos entender, depurar y extender sin tener que reescribirlo entero cada vez que agregamos una funcionalidad.

11. Adenda: correcciones hechas después del coloquio

Qué nos marcaron y cómo lo encaramos

Después del coloquio nos dimos cuenta de que teníamos dos problemas distintos mezclados: por un lado, cosas de implementación que había que corregir sí o sí; por otro, conclusiones del informe que estaban mal interpretadas. En esta reentrega tratamos de ordenar eso: entender bien cada observación, corregir código donde hacía falta.

Concurrencia, escalabilidad y throughput

En la primera versión habíamos dejado la idea de que el problema era usar un solo *event loop*. Eso estaba mal planteado. El modelo no bloqueante con un loop único sigue siendo válido para este tipo de servidor. Lo que sí nos limita hoy es el backend de multiplexación que usamos (`pselect`) y el límite de descriptores (`FD_SETSIZE`). Como cada conexión de túnel usa dos FDs, más algunos FDs fijos del proceso, no es realista sostener las cifras más altas que habíamos mencionado antes.

También corregimos cómo leer los resultados de stress: una conexión SOCKS exitosa (`REP=0x00`) no es lo mismo que "éxito HTTP". Son capas distintas.

Resolución DNS para FQDN

Otro punto importante fue FQDN. Antes, si un dominio resolvía a varias direcciones, el comportamiento no era robusto en todos los casos de fallo. Ahora el servidor intenta la lista completa de direcciones devuelta por `getaddrinfo`: si falla una, pasa a la siguiente hasta conectar o agotar opciones. Además, guardamos y liberamos bien esa estructura durante el ciclo de vida de la conexión para no dejar recursos colgados.

Señales y cierre limpio

Nos habían marcado que no teníamos manejo de `Ctrl-C`. Eso se corrigió agregando manejo de `SIGINT` y `SIGTERM`. El server ahora sale por un camino controlado: corta el loop principal y hace limpieza ordenada de los módulos (resolver, selector y sockets). También ignoramos `SIGPIPE` para evitar caídas bruscas cuando un socket ya fue cerrado del otro lado.

Protocolo de monitoreo y cliente

En la entrega anterior el protocolo de monitoreo existía, pero estaba poco formalizado. Eso lo ordenamos: dejamos definido formato de comandos y respuestas, y cómo se consultan métricas y comandos de configuración en runtime.

Además, implementamos un cliente propio de monitoreo en C (con modo interactivo y modo script), integrado al proceso de compilación. Eso cubre el requisito de no depender solo de `netcat` para operar el monitor.

Lecturas parciales en el monitor

Este fue un bug concreto: asumir que un `recv()` traía una línea completa. En TCP eso no está garantizado. Ahora el monitor usa un buffer acumulativo por conexión y procesa recién cuando encuentra delimitador de línea. Si entra un comando demasiado largo, responde error en vez de romperse.

Nuevo requisito (captura de credenciales)

Para esta entrega también sumamos el punto nuevo de captura de credenciales (POP3 y HTTP Basic Auth). La lógica quedó integrada al túnel: parseo incremental de `USER`/`PASS` para POP3 y de `Authorization: Basic` para HTTP. Lo capturado se guarda en un log separado (`credentials.log`) con timestamp, origen, destino, usuario y contraseña.

También dejamos claras las limitaciones esperables: tráfico cifrado (por ejemplo HTTPS sobre TLS) no se puede inspeccionar en contenido.