

# TRABAJO PRÁCTICO

## PROTOCOLOS DE COMUNICACIÓN

AUGUSTO CICCIOMESSERE - 62267

NICANOR PORTO - 62182

SIMON MARABI - 61626

IAN FRANCO TOGNETTI - 61215



## ÍNDICE

<b>1. INTRODUCCIÓN</b>	<b>3</b>
<b>2. DESCRIPCIÓN DETALLADA DE LOS PROTOCOLOS Y APLICACIONES DESARROLLADAS</b>	<b>3</b>
2.1. Arquitectura general del sistema	3
2.2. Protocolos soportados	4
2.3. Funcionalidades implementadas	4
<b>3. PROBLEMAS ENCONTRADOS DURANTE EL DISEÑO Y LA IMPLEMENTACIÓN</b>	<b>5</b>
3.1 Pruebas de stress iniciales insuficientes	5
3.2 Resolución de nombres bloqueante	5
3.3 Socket de management bloqueante dentro del selector	5
3.4 Manejo incorrecto de lecturas parciales y busy waiting	6
3.5 Corrupción de datos en transferencias sucesivas	6
3.6 Sniffer POP3 incompleto	6
<b>4. LIMITACIONES DE LA APLICACIÓN</b>	<b>6</b>
<b>5. POSIBLES EXTENSIONES</b>	<b>7</b>
<b>6. CONCLUSIONES</b>	<b>7</b>
<b>7. EJEMPLOS DE PRUEBA</b>	<b>8</b>
7.1. Prueba de conexión básica a través del proxy	8
7.2. Autenticación de usuario exitosa y fallida	8
7.3. Resolución por FQDN e IP	9
7.4. Monitoreo con el cliente administrativo	9
7.5. Sniffeeo de credenciales POP3	9
7.6. Pruebas de stress y rendimiento (ACTUALIZADO)	10
<b>8. GUÍA DE INSTALACIÓN DETALLADA Y PRECISA</b>	<b>10</b>
8.1 Requisitos	10
8.2 Instrucciones	11
8.2.1 Clonar el repositorio o descomprimir el paquete entregado:	11
8.2.2 Compilar el proyecto con make	11
8.2.3 Verificar que se hayan generado los binarios en ./bin/	11
<b>9. INSTRUCCIONES PARA LA CONFIGURACIÓN</b>	<b>11</b>
Gestión de usuarios	11
<b>10. EJEMPLOS DE CONFIGURACIÓN Y MONITOREO</b>	<b>11</b>
<b>11. DOCUMENTO DE DISEÑO DEL PROYECTO</b>	<b>12</b>
Arquitectura General	12
Estructura de carpetas	12
Flujo principal	13
11.1 Protocolo SOCKS5	13
11.2 Disectores POP3	15
11.3 Protocolo de gestión y monitoreo	15
11.4 Estabilidad de la ABI	15
<b>12. CONCLUSIONES</b>	<b>16</b>

# 1. INTRODUCCIÓN

El presente informe documenta el desarrollo del Trabajo Práctico Especial para la materia Protocolos de Comunicación. El objetivo principal del trabajo fue diseñar e implementar un servidor proxy conforme al protocolo SOCKS5, con soporte de autenticación mediante usuario y contraseña, y capacidades extendidas de monitoreo y administración.

Dicho protocolo fue desarrollado de forma robusta, escalable y eficiente, capaz de manejar múltiples conexiones simultáneas utilizando técnicas de I/O no bloqueante y multiplexación. Asimismo, el servidor incluye funcionalidades avanzadas como recolección de métricas en tiempo real, configuración dinámica sin reinicios, auditoría de accesos por usuario y monitoreo del tráfico POP3 para detectar credenciales.

En este documento se detalla la arquitectura general del sistema, las decisiones de diseño tomadas, los principales desafíos enfrentados durante la implementación, y las pruebas realizadas para validar el correcto funcionamiento y desempeño de la aplicación. A su vez, se proporciona una guía completa para la instalación, configuración y uso del sistema, y propuestas de mejoras.

Esta versión del informe amplía y complementa la entrega original, incorporando explícitamente los problemas detectados en la primera entrega, las correcciones realizadas, las decisiones de diseño adoptadas y las limitaciones remanentes.

## 2. DESCRIPCIÓN DETALLADA DE LOS PROTOCOLOS Y APLICACIONES DESARROLLADAS

### 2.1. Arquitectura general del sistema

El sistema desarrollado consiste en un servidor proxy SOCKS5 implementado en lenguaje C (estándar C11), con soporte para conexiones entrantes de múltiples clientes de forma simultánea y concurrente, mediante un único hilo de ejecución y utilizando I/O no bloqueante con `select()` como mecanismo de multiplexación.

El servidor cumple con los estándares definidos por los RFCs 1928 (protocolo SOCKS5) y 1929 (autenticación usuario/contraseña). Se diseñó para aceptar peticiones de conexión de clientes que desean acceder a servicios TCP externos, oficiando como intermediario entre el cliente y el servidor destino.

A su vez, se implementaron dos aplicaciones principales:

- **Servidor principal:** procesa las conexiones entrantes y maneja el flujo completo de datos entre cliente y servidor destino.
- **Cliente administrativo:** permite consultar métricas del servidor y modificar su configuración en tiempo de ejecución sin necesidad de reiniciar el proceso

## 2.2. Protocolos soportados

- **SOCKS5 (RFC1928)**: Protocolo de proxy que permite a los clientes establecer conexiones TCP a través del servidor. Soporta direccionamiento por IP (IPv4 e IPv6) y por nombres de dominio (FQDN).
- **Autenticación (RFC1929)**: Se exige autenticación de usuario y contraseña en la fase de negociación. El servidor mantiene una lista de credenciales válidas y verifica cada intento de acceso.
- **Protocolo interno de administración y monitoreo**: Se definió un protocolo propietario simple, basado en mensajes de texto, que permite obtener estadísticas del servidor (como cantidad de conexiones activas o bytes transferidos) y actualizar configuraciones como los usuarios habilitados o parámetros operativos.

## 2.3. Funcionalidades implementadas

- Manejo de múltiples clientes (al menos 500 conexiones activas).
- Autenticación obligatoria de usuarios antes de permitir el acceso.
- Soporte de IPv4, IPv6 y FQDNs.
- Manejo robusto de resolución DNS (si una IP falla, intenta con las restantes).
- Registro de accesos por usuario, incluyendo fecha y hora de cada conexión.
- Módulo de recolección de métricas, incluyendo:
  - Conexiones históricas totales.
  - Conexiones simultáneas actuales.
  - Cantidad total de bytes transferidos.
- Modificación dinámica de la configuración en tiempo de ejecución.
- Monitoreo de tráfico POP3, con log de credenciales interceptadas (usuarios y contraseñas).
- Cliente administrativo que permite interactuar con el servidor usando el protocolo interno.

### 3. PROBLEMAS ENCONTRADOS DURANTE EL DISEÑO Y LA IMPLEMENTACIÓN

Durante el desarrollo del sistema se presentaron diversos desafíos técnicos que requirieron análisis, pruebas y decisiones de diseño cuidadosas para asegurar el correcto funcionamiento del proxy. Además, tuvimos varios errores presentados en la primera entrega del trabajo, por lo que también haremos especial énfasis en detallar esos inconvenientes.

#### 3.1 Pruebas de stress iniciales insuficientes

En la primera entrega, las pruebas de performance se basaban en un cliente que únicamente establecía conexiones SOCKS5, realizaba el greeting y finalizaba. De esta forma nunca se evaluaba la transferencia real de datos ni la simultaneidad efectiva de sockets cliente–origen y por ese motivo, la métrica obtenida no representaba el comportamiento del servidor bajo carga real, ya que no existía relay de datos entre sockets.

Para corregir esto se desarrolló un cliente de estrés específico que ejecuta el flujo completo del protocolo (GREETING, AUTH, CONNECT) y luego realiza transferencia efectiva de datos hacia un servidor remoto controlado. De esta forma se validó el manejo simultáneo de 500 conexiones activas haciendo relay de tráfico real.

#### 3.2 Resolución de nombres bloqueante

Otro problema fue que la resolución DNS mediante `getaddrinfo()` se realizaba en el hilo principal del servidor, bloqueando el loop de multiplexación cuando la resolución demoraba. Esto generaba que durante la resolución de nombres, el servidor dejaba de aceptar nuevas conexiones y de avanzar con las existentes.

Para solucionar este problema, la resolución de nombres (`getaddrinfo()`) y el `connect()` se delegan a threads auxiliares mediante la función `resolver_thread()`. Esta función ejecuta `socks5_connect_and_reply()` en un contexto separado, permitiendo que el hilo principal continúe atendiendo otras conexiones. Una vez completada la operación, el thread escribe el resultado (`dns_result_t`) en un pipe no bloqueante. El hilo principal detecta estos resultados mediante `select()` sobre el descriptor de lectura del pipe, integrándolos al flujo normal de eventos sin bloquear el loop de multiplexación.

#### 3.3 Socket de management bloqueante dentro del selector

El socket de administración estaba configurado como bloqueante pero registrado dentro del selector principal. Por lo tanto, el handler podía bloquearse al obtener métricas protegidas por mutex y durante envíos largos, afectando el procesamiento de conexiones SOCKS5.

Para solucionar este problema, el manejo del socket de administración fue rediseñado para operar de forma completamente no bloqueante dentro del selector principal. Se configura el socket de escucha con `set_nonblocking()` y cada cliente de management se gestiona

mediante una estructura `mgmt_client_t` con máquina de estados. Las operaciones de lectura y escritura utilizan `recv()` y `send()` no bloqueantes, manejando `EAGAIN/EWOULDBLOCK` para evitar bloqueos. De esta forma, el servidor de management se integra completamente en el loop de `select()` sin afectar el procesamiento de conexiones `SOCKS5`.

### 3.4 Manejo incorrecto de lecturas parciales y busy waiting

Otro punto que tuvimos inconveniente fue que no se manejaban correctamente las lecturas parciales. Ante clientes que no enviaban datos, el servidor podía entrar en un loop activo (`busy waiting`). Esto impactaba en el uso innecesario de CPU y degradación del rendimiento bajo ciertas condiciones.

Para solucionar este problema, se implementó un manejo completo de lecturas y escrituras parciales mediante buffers con seguimiento de posición. Las funciones `try_read_into()` y `try_flush()` operan de forma no bloqueante: si `recv()` o `send()` retornan `EAGAIN/EWOULDBLOCK`, la operación se suspende y el progreso se preserva en variables de estado (`*len`, `*sent`). Esto permite retomar la transferencia en la próxima iteración cuando `select()` indique disponibilidad. Adicionalmente, el loop principal utiliza `select()` con un timeout configurable, asegurando que el servidor solo procese cuando hay eventos reales y evitando ciclos de CPU innecesarios (`busy waiting`).

### 3.5 Corrupción de datos en transferencias sucesivas

El problema detectado era que durante transferencias grandes, el primer request funcionaba correctamente pero los siguientes abortaban o devolvían datos corruptos. Por lo tanto, la integridad de los datos transferidos no estaba garantizada.

Para solucionar este problema, se reestructuró el sistema de buffers de relay utilizando un esquema de seguimiento completo de estado. Cada conexión emplea buffers dinámicos (`c2r_buf`, `r2c_buf`) con contadores independientes para datos pendientes (`*_len`) y bytes ya transmitidos (`*_sent`). La función `try_read_into()` evita sobreescrituras al no cargar nuevos datos mientras existan datos sin enviar. La función `try_flush()` transmite desde la posición correcta y solo reinicia los contadores cuando el envío finaliza. Este diseño asegura que las escrituras parciales no pierdan información y que los buffers no se corrompan entre transferencias consecutivas.

### 3.6 Sniffer POP3 incompleto

Se reescribió el sniffer POP3 integrándolo en el flujo del relay. El módulo ahora detecta tanto `USER/PASS` como `AUTH PLAIN`, decodificando el blob Base64 para extraer las credenciales. Una máquina de estados acumula los datos, y al capturar un par usuario/contraseña los registra en `pop3_credentials.log` con timestamp e IP de origen.

## 4. LIMITACIONES DE LA APLICACIÓN

Si bien el servidor proxy desarrollado cumple con todos los requerimientos funcionales y no funcionales, existen algunas limitaciones propias del diseño actual que podrían ser abordadas en versiones futuras:

- **Escalabilidad limitada por `select()`:** El uso de `select()` como mecanismo de multiplexación impone una barrera en cuanto a la cantidad máxima de descriptores de archivo manejables simultáneamente, aunque se cumple con el mínimo requerido de 500 conexiones.
- **Persistencia de métricas:** Las métricas recolectadas (como cantidad de conexiones históricas o bytes transferidos) son volátiles y se pierden al reiniciar el servidor, limitando el análisis histórico.
- **Falta de cifrado en el canal de administración:** El protocolo de administración se basa en texto plano, sin cifrado ni autenticación adicional, lo que podría representar un riesgo si se utiliza en entornos inseguros o redes compartidas.
- **Soporte limitado a protocolos sniffables:** El monitoreo de credenciales está limitado únicamente al protocolo POP3. Otros protocolos inseguros (como HTTP, FTP o IMAP sin TLS) no son analizados por el sistema.
- **Gestión de usuarios básica:** La administración de usuarios habilitados para acceder al proxy se realiza mediante archivos de configuración sin herramientas externas ni integración con sistemas de autenticación centralizados.

## 5. POSIBLES EXTENSIONES

Existen diversas mejoras y extensiones posibles que podrían incorporarse a futuras versiones del sistema para ampliar sus capacidades, facilitar su administración o mejorar su desempeño. Algunas de las más relevantes son:

- **Migración a `epoll` o `kqueue`:** Sustituir `select()` por mecanismos más eficientes de multiplexación permitiría escalar el servidor a miles de conexiones concurrentes, reduciendo la carga sobre la CPU y mejorando la latencia.
- **Persistencia de métricas y logs:** Integrar una base de datos o sistema de almacenamiento persistente permitiría mantener estadísticas históricas y trazabilidad completa de accesos incluso después de reinicios del servidor.
- **Cifrado del canal de administración:** Incorporar TLS al protocolo de administración permitiría asegurar la confidencialidad e integridad de las comunicaciones entre el cliente administrativo y el servidor.

- **Soporte para otros protocolos inseguros:** Extender el módulo de sniffeo para detectar credenciales en protocolos adicionales como HTTP básico, FTP o IMAP sin cifrado.
- **Limitación de uso por usuario:** Implementar cuotas o límites por usuario (por ejemplo, máximo de conexiones simultáneas o cantidad de datos transferidos) permitiría un control más fino del uso del proxy.

## 6. CONCLUSIONES

El desarrollo de este servidor proxy SOCKS5 representó un desafío integral que combinó conceptos teóricos de protocolos de comunicación con una implementación práctica exigente en C, respetando los estándares definidos por los RFCs y los requisitos de rendimiento y escalabilidad planteados por la cátedra.

A lo largo del trabajo se lograron implementar todas las funcionalidades obligatorias y opcionales, incluyendo autenticación, multiplexación no bloqueante, configuración dinámica, recolección de métricas y monitoreo de tráfico POP3. Además, se diseñó un protocolo interno para administración y un cliente específico para facilitar el monitoreo del sistema.

El proyecto permitió afianzar conocimientos clave sobre programación de sockets, manejo eficiente de múltiples conexiones, diseño de protocolos, gestión de estado y control de errores en sistemas concurrentes. Asimismo, se desarrolló una estructura modular y extensible, con miras a posibles mejoras futuras.

Si bien se identificaron ciertas limitaciones, como la ausencia de cifrado en el canal de administración o la falta de persistencia en las métricas, el sistema demuestra un funcionamiento sólido, robusto y alineado con los objetivos del trabajo práctico.

En resumen, el proyecto cumplió con los requerimientos planteados, ofreciendo una solución funcional, completa y extensible para la implementación de un servidor proxy orientado a entornos de alto rendimiento y administración dinámica.

### 6.1 Conclusiones de segunda entrega

Tras las correcciones recibidas y el trabajo realizado para satisfacerlas, consideramos que se atendió lo más grave de este trabajo, que era que este protocolo resultaba bloqueante, además de errores relacionados al busy waiting y corrupción de datos.

Para esta reentrega se mejoró sustancialmente la robustez y el realismo de las pruebas realizadas, así como también eliminar puntos de bloqueo y errores sutiles de concurrencia. Resultando en un trabajo que consideramos más acorde y que va más de la mano con la idea del mismo.

## 7. EJEMPLOS DE PRUEBA

A continuación, se detallan algunos ejemplos de pruebas realizadas para verificar el correcto funcionamiento del sistema proxy SOCKS5, tanto desde el punto de vista funcional como del monitoreo y la administración.

### 7.1. Prueba de conexión básica a través del proxy

1. Iniciar el servidor SOCKS5:

```
./bin/socks5
```

2. Ejecutar una conexión de prueba desde un cliente curl configurado para usar el proxy:

```
curl --socks5 127.0.0.1:1080 --proxy-user usuario:clave https://www.google.com
```

2. Ejecutar una conexión de prueba desde un cliente ncat configurado para usar el proxy:

```
ncat --proxy 127.0.0.1:1080 --proxy-type socks5 --proxy-auth user:pass protos.foo 80
```

Resultado esperado: se recibe correctamente el contenido HTML de `example.com` a través del proxy, probado con curl y ncat.

### 7.2. Autenticación de usuario exitosa y fallida

1. Usuario válido:

```
curl --proxy-user user1:pass1 --socks5 127.0.0.1:1080 https://www.google.com
```

2. Usuario inválido:

```
curl --proxy-user wrong:wrong --socks5 127.0.0.1:1080 https://www.google.com
```

Resultado esperado: en el primer caso, acceso exitoso. En el segundo caso, rechazo de conexión con código de error SOCKS5.

### 7.3. Resolución por FQDN e IP

1. Por nombre de dominio:

```
curl --proxy-user user:pass --socks5-hostname 127.0.0.1:1080 https://www.google.com
```

2. Por dirección IP directa:

```
curl --proxy-user user:pass --socks5 127.0.0.1:1080  
http://172.217.173.238
```

Resultado esperado: ambas conexiones deben completarse correctamente.

## 7.4. Monitoreo con el cliente administrativo

1. Consulta de métricas y configuración del cliente :

```
./bin/client -s  
  
./bin/client -c
```

Muestra: cantidad de conexiones totales, activas, y bytes transferidos.

2. Modificación de parámetros:

```
./bin/client -m <número>  
./bin/client -b <tamaño del buffer>  
./bin/client -t <timeout>
```

Agrega un nuevo usuario al sistema en tiempo de ejecución, sin reiniciar el servidor.

## 7.5. Sniffeeo de credenciales POP3

1. Ejecutar un cliente POP3 configurado sin TLS, a través del proxy.

```
ncat --proxy 127.0.0.1:1080 --proxy-type socks5 --proxy-auth user:password  
127.0.0.1 110
```

2. Observar el log generado:

```
cat pop3_credentials.log
```

Resultado esperado: el archivo contiene usuario y contraseña interceptados del flujo POP3.

## 7.6. Pruebas de stress y rendimiento (ACTUALIZADO)

Con el objetivo de evaluar la escalabilidad y el rendimiento del servidor SOCKS5 implementado, se diseñó una prueba de stress que simula múltiples clientes concurrentes intentando conectarse de manera simultánea.

Se utilizó un script de benchmark asíncrono que genera miles de conexiones simultáneas al servidor y mide la tasa de handshakes completados por segundo

## 1. Compilar:

`make stress` (Nota: por default se utiliza el puerto 1080, si se desea modificar utilizar: `make stress STRESS_PORT=NR0_PUERTO STRESS_SINK_PORT=NR0_PUERTO`)

Las pruebas de stress fueron rediseñadas para reflejar carga real. Se ejecutaron escenarios con 500 conexiones concurrentes activas, cada una transfiriendo 1 MB de datos a través del proxy, verificando:

- simultaneidad efectiva,
- ausencia de corrupción de datos,
- estabilidad del throughput.

Los resultados muestran que el servidor mantiene correctamente todas las conexiones y completa la transferencia total sin errores.

# 8. GUÍA DE INSTALACIÓN DETALLADA Y PRECISA

## 8.1 Requisitos

- Sistema Operativo: Linux y MacOS.
- Dependencias: make, gcc (soporte para C11), libc, libpthread, nc o ncat, curl.
- Espacio en disco estimado: 2 MB

## 8.2 Instrucciones

### 8.2.1 Clonar el repositorio (branch main) o descomprimir el paquete entregado:

```
git clone https://github.com/ACicciomessere/TP-Protos.git
```

### 8.2.2 Compilar el proyecto con make

```
make clean
make
```

### 8.2.3 Verificar que se hayan generado los binarios en ./bin/

```
socks5: servidor principal
client: cliente administrativo
```

# 9. INSTRUCCIONES PARA LA CONFIGURACIÓN

La configuración del sistema se realiza de forma dinámica mediante el cliente administrativo, sin necesidad de reiniciar el servidor. Además, ciertos parámetros pueden configurarse al iniciar el servidor.

Ejecución del servidor

```
./bin/socks5 [-p puerto_proxy]
```

-p: puerto de escucha del proxy SOCKS5 (por defecto: 1080)

### **Gestión de usuarios**

Agregar un usuario:

```
./bin/client -u usuario:contraseña
```

Borrar un usuario:

```
./bin/client -d usuario
```

Usuarios Configurados:

```
./bin/client -l
```

## **10. EJEMPLOS DE CONFIGURACIÓN Y MONITOREO**

Setear máximo número de clientes.

```
./bin/client -m <número>
```

Setear tamaño del buffer

```
./bin/client -b <tamaño del buffer>
```

Setear el timeout

```
./bin/client -t <timeout>
```

Ver configuración actual del servidor

```
./bin/client -c
```

Versión del servidor

```
./bin/client -v
```

# 11. DOCUMENTO DE DISEÑO DEL PROYECTO

## 11.1 Primera entrega del proyecto

### Arquitectura General

El proyecto se estructura en tres componentes principales:

- **Servidor SOCKS5 (main.c):** Servidor proxy que procesa clientes concurrentes usando select() con I/O 100% no bloqueante. Soporta IPv4, IPv6 y resolución FQDN asíncrona vía threads.
- **Módulo POP3 Sniffer:** Intercepta y logea credenciales del protocolo POP3, incluyendo soporte para AUTH PLAIN con decodificación Base64.
- **Cliente Administrativo:** Permite interactuar con el servidor para monitoreo de estadísticas, gestión de usuarios y cambios de configuración en tiempo real.

### Estructura de carpetas

```
src/
├── main.c           # Servidor principal SOCKS5
├── client.c         # Cliente de gestión remota
├── shared.c/.h      # Funciones compartidas
├── core/
│   ├── buffer.c/.h  # Manejo de buffers
│   ├── selector.c/.h # Multiplexor I/O
│   └── stm.c/.h     # Máquina de estados
├── docs/
│   ├── socks5d.8
│   └── Informe.pdf
├── protocols/
│   ├── socks5/
│   │   └── socks5.c/.h # Protocolo SOCKS5
│   └── pop3/
│       └── pop3_sniffer.c/.h # Sniffer POP3
├── utils/
│   ├── args.c/.h    # Parser de argumentos
│   ├── logger.c/.h  # Sistema de logging
│   └── util.c/.h     # Funciones auxiliares
└── tests/
    ├── pop3_test.c   # Tests POP3
    └── socks5_tests.c # Tests SOCKS5
```

### Flujo principal

1. El servidor escucha conexiones entrantes por el puerto proxy.

2. Valida usuario/contraseña según RFC1929.
3. Establece conexión TCP a destino (IPv4, IPv6, FQDN).
4. Si el protocolo detectado es POP3, se interceptan credenciales.
5. En paralelo, escucha peticiones administrativas por el puerto designado.

## 11.2 Segunda entrega del proyecto

### Estructura de carpetas

TP-Protos/

```

├── Makefile          # Build principal
├── Makefile.inc      # Flags condicionales (Linux/macOS)
├── README.md
├── auth.db           # Base de datos de usuarios
├── pop3_credentials.log # Log de credenciales POP3 interceptadas
├── metrics.log       # Métricas del servidor
├── bin/              # Binarios compilados
├── obj/              # Archivos objeto
├── docs/             # Archivos relevantes al proyecto
├── tools/
│   ├── plot_stress.c # Visualización de tests de estrés
│   └── sink_server.c # Servidor sink para testing
├── src/
│   ├── main.c        # Servidor principal SOCKS5 (loop no bloqueante)
│   ├── client.c      # Cliente de gestión remota
│   ├── shared.c/.h    # Protocolo de gestión + memoria compartida
│   ├── core/
│   │   ├── buffer.c/.h # Manejo de buffers circulares
│   │   ├── selector.c/.h # Multiplexor I/O (select wrapper)
│   │   └── stm.c/.h    # Máquina de estados finitos
│   ├── protocols/
│   │   ├── socks5/
│   │   │   └── socks5.c/.h # Implementación protocolo SOCKS5
│   │   └── pop3/
│   │       └── pop3_sniffer.c/.h # Sniffer POP3 (USER/PASS/AUTH PLAIN)
│   ├── utils/
│   │   ├── args.c/.h    # Parser de argumentos CLI
│   │   ├── logger.c/.h  # Sistema de logging con niveles
│   │   └── util.c/.h    # Funciones auxiliares
│   ├── tests/
│   │   ├── pop3_test.c  # Tests unitarios POP3
│   │   ├── socks5_tests.c # Tests unitarios SOCKS5
│   │   └── stress_client.c # Cliente de pruebas de carga
│   └── docs/
│       └── socks5d.8     # Man page del servidor

```

## Flujo principal

1. El servidor escucha conexiones entrantes por el puerto proxy (IPv4 e IPv6).
2. Al aceptar una conexión, se configura como non-blocking y pasa a STATE\_GREETING.
3. Lee los métodos de autenticación ofrecidos por el cliente y responde con el método seleccionado (STATE\_GREETING\_REPLY).
4. Si se requiere autenticación, valida usuario/contraseña según RFC1929 (STATE\_AUTH → STATE\_AUTH\_REPLY).
5. Parsea el request CONNECT con destino IPv4, IPv6 o FQDN (STATE\_REQUEST).
6. Lanza un thread asíncrono para resolución DNS + connect al destino (STATE\_CONNECTING).
7. El thread notifica al loop principal vía pipe (dns\_pipe\_fds) cuando la conexión está lista.
8. Entra en modo relay bidireccional (STATE\_RELAYING) usando buffers circulares.
9. Si el puerto destino es 110 (POP3), activa el sniffer para interceptar credenciales.
10. Las credenciales POP3 detectadas (USER/PASS, AUTH PLAIN) se loguean a pop3\_credentials.log.
11. En paralelo, el servidor acepta y procesa peticiones administrativas de forma no bloqueante por el puerto de gestión.

### 11.2.1 Protocolo SOCKS5

El servidor implementa el protocolo SOCKS5 conforme a lo especificado en los RFC 1928 y 1929. Se soporta autenticación obligatoria mediante usuario/contraseña (método 0x02) y el comando CONNECT para el establecimiento de conexiones TCP hacia destinos IPv4, IPv6 o FQDN.

El flujo completo del protocolo se divide en las siguientes etapas:

#### Greeting

El cliente inicia la comunicación enviando la versión del protocolo y los métodos de autenticación soportados:

VER | NMETHODS | METHODS

El servidor responde seleccionando el método de autenticación a utilizar.

#### Autenticación (RFC 1929)

Si el método seleccionado es usuario/contraseña, el cliente envía:

VER | ULEN | UNAME | PLEN | PASSWD

El servidor valida las credenciales contra su configuración interna y responde indicando éxito o fallo.

### **Request CONNECT**

Una vez autenticado, el cliente solicita el establecimiento de una conexión TCP enviando:

VER | CMD | RSV | ATYP | DST.ADDR | DST.PORT

Solo se soporta el comando CONNECT (CMD = 0x01).

### **Relay de datos**

Tras una respuesta exitosa, el servidor actúa como intermediario entre el cliente y el destino, relayando tráfico bidireccional de forma transparente. El flujo de datos es multiplexado utilizando select() y contabilizado dentro de las métricas del sistema.

Si el destino corresponde al puerto TCP 110 y los disectores están habilitados, los payloads son inspeccionados por el sniffer POP3.

### **Opciones y parámetros relevantes**

- **Autenticación:** se pueden configurar hasta 10 usuarios al iniciar el servidor mediante la opción -u user:pass. Los usuarios también pueden agregarse o eliminarse dinámicamente a través del protocolo de gestión (CMD\_ADD\_USER / CMD\_DEL\_USER).
- **Timeout de resolución y conexión:** configurable dinámicamente mediante el comando CMD\_SET\_TIMEOUT, expresado en milisegundos. El valor por defecto es 10 segundos.
- **Buffers de relay:** el comando CMD\_SET\_BUFFER permite ajustar en tiempo de ejecución el tamaño del buffer circular utilizado para el relay de datos, con valores entre 512 y 65536 bytes. El cambio se aplica en caliente a todas las conexiones activas.
- **Disectores:** pueden habilitarse o deshabilitarse dinámicamente mediante los comandos CMD\_ENABLE\_DISSECTORS y CMD\_DISABLE\_DISSECTORS.

### **Respuestas de error (REP)**

El servidor utiliza los códigos oficiales definidos por SOCKS5 para reportar errores al cliente:

- 0x01: fallo general.
- 0x02: regla de red no permitida.
- 0x03: red inalcanzable.

- 0x04: host inalcanzable (utilizado cuando falla la resolución mediante `getaddrinfo()`).
- 0x05: conexión rechazada.
- 0x06: TTL expirado.
- 0x07: comando no soportado (cuando CMD no es CONNECT).
- 0x08: tipo de dirección no soportado.

### Estado interno

Cada conexión se maneja mediante una máquina de estados finita con la siguiente secuencia:

STATE\_GREETING → STATE\_AUTH → STATE\_REQUEST → STATE\_RELAYING

Los descriptors de archivo se registran dinámicamente en el `select()` principal, con intereses de lectura y escritura según la presencia de datos pendientes en los buffers de cada conexión.

#### 11.2.2 Disectores POP3

El sniffer POP3 inspecciona exclusivamente sesiones cuyo destino corresponde al puerto TCP 110. Durante el relay de datos, se analizan los comandos del protocolo POP3 y se detectan tanto los mecanismos clásicos USER/PASS como el método de autenticación AUTH PLAIN, decodificando el payload Base64 cuando corresponde.

Cuando se capturan credenciales, estas se registran en el archivo `pop3_credentials.log` y se emite una traza informativa en `metrics.log`, indicando usuario, contraseña y contexto de la conexión.

Los disectores se encuentran habilitados por defecto. Si el servidor se inicia con la opción `-N`, quedan deshabilitados de forma permanente. En tiempo de ejecución pueden habilitarse o deshabilitarse mediante el protocolo de gestión utilizando los comandos `CMD_ENABLE_DISSECTORS` y `CMD_DISABLE_DISSECTORS`. Cada cambio de estado queda explícitamente registrado en los logs del sistema.

#### 11.2.3 Protocolo de gestión y monitoreo

El sistema implementa un protocolo propietario de gestión y monitoreo servido sobre TCP. La API utiliza estructuras binarias de tamaño fijo definidas en `shared.h`. Cada solicitud consiste en el envío completo de una estructura `mgmt_message_t`, seguida por la recepción de la estructura de respuesta asociada al comando solicitado.

Los comandos soportados incluyen, entre otros:

- CMD\_ADD\_USER/CMD\_DEL\_USER, con respuesta `mgmt_simple_response_t`.
- CMD\_LIST\_USERS, con respuesta `mgmt_users_response_t`.
- CMD\_STATS, con respuesta `mgmt_stats_response_t`.
- CMD\_SET\_TIMEOUT.
- CMD\_SET\_BUFFER.
- CMD\_SET\_MAX\_CLIENTS.
- CMD\_ENABLE\_DISSECTORS / CMD\_DISABLE\_DISSECTORS.
- CMD\_RELOAD\_CONFIG.
- CMD\_GET\_CONFIG.

Todas las solicitudes utilizan el formato base `mgmt_message_t` y admiten únicamente caracteres ASCII. Los campos se rellenan con ceros cuando no se utilizan. El campo `username` se reutiliza para pasar argumentos numéricos en formato decimal cuando el comando lo requiere (por ejemplo, tamaño de buffer o timeout).

Las respuestas se envían mediante `send_all()` y se reciben mediante `recv_all()`, garantizando la transmisión completa de las estructuras. Cada conexión de gestión es atendida en un hilo dedicado y se cierra una vez procesado el comando solicitado.

#### 11.2.4 Estabilidad de la ABI

Las estructuras definidas en `shared.h` constituyen la ABI (Application Binary Interface) del protocolo de gestión. Cualquier modificación en su layout debe reflejarse tanto en este documento como en los clientes que consumen la API.

El layout actual de la estructura base es:

```
mgmt_message_t {
    mgmt_command_t command;    // enum (32 bits)
    char username[64];
    char password[64];
}
```

Las respuestas reutilizan la estructura `mgmt_simple_response_t` (campo de éxito y mensaje descriptivo) o estructuras específicas como `mgmt_stats_response_t` y `mgmt_config_response_t`, según el comando ejecutado.