72.07 Protocolos de Comunicación

Trabajo Práctico Especial: Servidor Proxy para el protocolo SOCKSv5 [RFC1928]

Alumnos

Román Berruti - 63533

Lautaro Bonseñor - 62842

Toribio Viton Sconza - 64275

Mateo Hernán Cornejo - 62722

Docentes

Garberoglio, Marcelo Fabio

Kulesz, Sebastián

Axt Roberto Oscar, Roberto Oscar

Stupenengo Faus, Hugo Javier

Grupo 2

18/12/2025

Índice

1. Descripción detallada de los protocolos y aplicaciones desarrolladas

1.1. Introducción al proyecto

El presente trabajo práctico consiste en la implementación de un servidor proxy para el protocolo SOCKSv5, según lo especificado en el RFC 1928. El proxy actúa como intermediario entre clientes y servidores de destino, permitiendo a los clientes establecer conexiones TCP a través del mismo de manera transparente.

El sistema desarrollado está compuesto por dos aplicaciones principales:

1. Servidor Proxy SOCKS5: Implementa el protocolo SOCKSv5 completo, incluyendo autenticación usuario/contraseña (RFC 1929), y expone un puerto adicional para administración y monitoreo.

2. Cliente de Configuración/Monitoreo: Permite administrar el servidor en tiempo de ejecución mediante un protocolo de texto propietario.

1.2. Protocolo SOCKSv5 (RFC 1928)

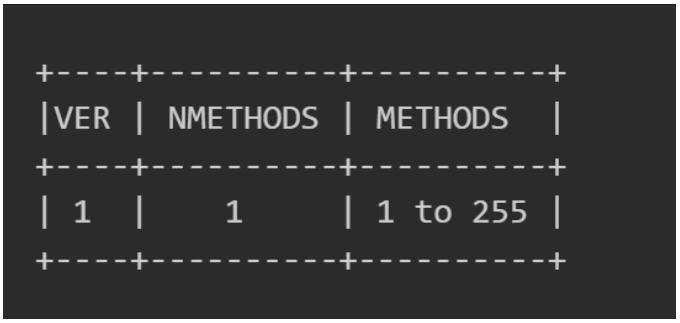
1.2.1. Descripción general del protocolo

SOCKS (Socket Secure) versión 5 es un protocolo de Internet que permite a un cliente establecer conexiones TCP a través de un servidor proxy. A diferencia de los proxies HTTP, SOCKS opera a un nivel más bajo (capa de transporte) y es agnóstico al protocolo de aplicación.

1.2.2. Proceso de negociación (Fase HELLO)

La comunicación inicia con una fase de negociación donde el cliente indica qué métodos de autenticación soporta:

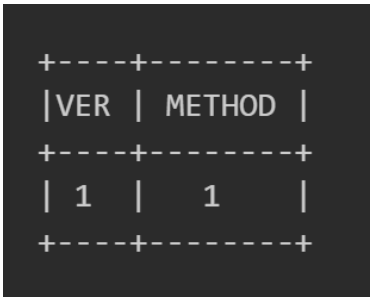
Mensaje del cliente:



- VER: Versión del protocolo (0x05 para SOCKSv5)

- NMETHODS: Número de métodos de autenticación soportados
- METHODS: Lista de métodos (0x00 = sin autenticación, 0x02 = usuario/contraseña)

Respuesta del servidor:

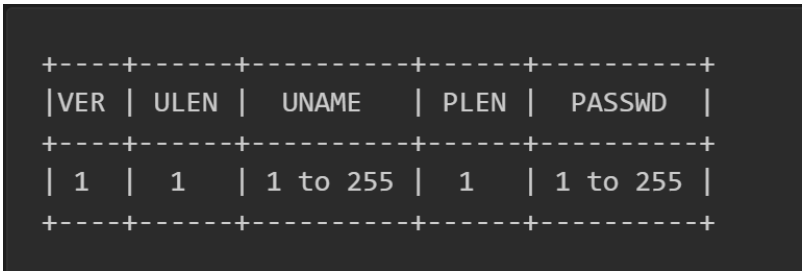


Nuestra implementación selecciona automáticamente el método 0x02 (autenticación) si hay usuarios configurados en el sistema, o 0x00 (sin autenticación) en caso contrario.

1.2.3. Autenticación usuario/contraseña (RFC 1929)

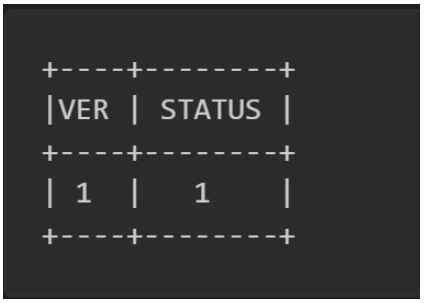
Si el servidor selecciona el método 0x02, el cliente debe enviar credenciales:

Mensaje del cliente:



- VER: Versión del subnegociación (0x01)
- ULEN: Longitud del nombre de usuario
- UNAME: Nombre de usuario
- PLEN: Longitud de la contraseña
- PASSWD: Contraseña

Respuesta del servidor:



- STATUS: 0x00 indica éxito, cualquier otro valor indica fallo

1.2.4. Fase de solicitud de conexión (REQUEST)

Una vez autenticado (o si no se requiere autenticación), el cliente envía la solicitud de conexión:

Mensaje del cliente:

```
+----+-----+-------+------+----------+----------+
|VER | CMD |  RSV  | ATYP | DST.ADDR | DST.PORT |
+----+-----+-------+------+----------+----------+
| 1  |  1  | X'00' |  1   | Variable |    2     |
+----+-----+-------+------+----------+----------+
```

- VER: Versión (0x05)
- CMD: Comando (0x01 = CONNECT, único soportado)
- RSV: Reservado (0x00)
- ATYP: Tipo de dirección
    - 0x01: IPv4 (4 bytes)
    - 0x03: Nombre de dominio (1 byte longitud + dominio)
    - 0x04: IPv6 (16 bytes)
- DST.ADDR: Dirección de destino
- DST.PORT: Puerto de destino (2 bytes, network byte order)

Respuesta del servidor:

```
+----+-----+-------+------+----------+----------+
|VER | REP |  RSV  | ATYP | BND.ADDR | BND.PORT |
+----+-----+-------+------+----------+----------+
| 1  |  1  | X'00' |  1   | Variable |    2     |
+----+-----+-------+------+----------+----------+
```

- REP: Código de respuesta
    - 0x00: Éxito
    - 0x01: Fallo general del servidor SOCKS
    - 0x03: Red inalcanzable
    - 0x04: Host inalcanzable
    - 0x05: Conexión rechazada
    - 0x07: Comando no soportado
    - 0x08: Tipo de dirección no soportado

1.2.5. Fase de relay de datos (COPY)

Una vez establecida la conexión, el proxy actúa como un túnel transparente, copiando datos bidireccionalmente entre el cliente y el servidor de destino. Esta fase continúa hasta que cualquiera de las partes cierra la conexión.

1.3. Protocolo de Configuración y Monitoreo

1.3.1. Diseño del protocolo

Diseñamos un protocolo de texto plano basado en TCP que permite administrar el servidor en tiempo de ejecución. Las características principales son:

- Puerto por defecto: 8080
- Dirección por defecto: 127.0.0.1 (solo localhost por seguridad)
- Formato: Texto plano, líneas terminadas en `\n`
- Codificación: ASCII/UTF-8

1.3.2. Formato de mensajes

Comandos del cliente:

```
<COMANDO> [<ARG1>] [<ARG2>] ...\n
```

Respuestas exitosas:

- Consultas: datos en formato “key=value” o texto formateado
- Modificaciones: “OK <mensaje>\n”

Respuestas de error:
ERR <mensaje de error>\n

1.3.3. Comandos implementados

STATS: Muestra estadísticas completas del servidor
USERS: Muestra cantidad de usuarios registrados
LISTUSERS: Lista todos los nombres de usuario
ADDUSER <user> <pass>: Agrega o actualiza un usuario
DELUSER <user>: Elimina un usuario
CREDS: Muestra estadísticas del monitor de credenciales
HELP: Muestra ayuda de comandos
QUIT: Cierra la conexión

Ejemplo de respuesta STATS:

```
=== SOCKS5 PROXY METRICS ===

Server Status:
  uptime=0d 2h 15m 42s

Connections:
  total=142
  current=5
  max_concurrent=23
  failed=3

Data Transfer:
  bytes_sent=1048576
  bytes_received=524288
  bytes_total=1572864

Authentication:
  success=140
  failed=2
```

1.4. Servidor Proxy SOCKS5

1.4.1. Arquitectura general

El servidor implementa una arquitectura basada en eventos (event-driven) con las siguientes características:

- Single-threaded para I/O: Todo el manejo de I/O ocurre en un único hilo principal
- Multiplexación con epoll: Uso de la API epoll de Linux para manejar múltiples conexiones concurrentemente
- Threads auxiliares para DNS: Resolución de nombres de dominio en threads separados para no bloquear el hilo principal
- Máquina de estados por conexión: Cada conexión se modela como una máquina de estados finita

1.4.2. Decisión de diseño: epoll vs select

Una de las decisiones de diseño más importantes fue utilizar epoll en lugar de select para la multiplexación de I/O. Las razones fueron:

Ventajas de epoll sobre select:

1. Escalabilidad O(1) vs O(n):
   - select debe iterar sobre todos los file descriptors registrados en cada llamada
   - epoll solo retorna los file descriptors que tienen eventos pendientes

- Para 500+ conexiones concurrentes, esta diferencia es crítica

2. Sin límite artificial de file descriptors:
   - select tiene un límite de `FD_SETSIZE` (típicamente 1024)
   - epoll no tiene límite práctico (solo memoria del sistema)

3. Estado persistente:
   - select requiere reconstruir los conjuntos de fd en cada llamada
   - epoll mantiene estado entre llamadas, reduciendo overhead

4. Edge-triggered disponible:
   - epoll permite modo edge-triggered para mayor eficiencia
   - (En nuestra implementación usamos level-triggered por simplicidad)

Implementación:

```
// Creación del descriptor epoll
s->epoll_fd = epoll_create1(EPOLL_CLOEXEC);

// Registro de file descriptors
struct epoll_event event;
event.events = EPOLLIN | EPOLLOUT;  // Según interés
event.data.ptr = &s->fds[fd];
epoll_ctl(s->epoll_fd, EPOLL_CTL_ADD, fd, &event);

// Event loop
int n = epoll_wait(s->epoll_fd, events, MAX_EVENTS, timeout);
for (int i = 0; i < n; i++) {
    // Solo procesamos los FDs con eventos
}
```

1.4.3. Manejo de concurrencia

El servidor maneja múltiples conexiones concurrentes mediante:

1. Selector (epoll): Registra todos los sockets y notifica cuando hay eventos
2. Sockets no bloqueantes: Todas las operaciones de I/O son non-blocking
3. Buffers por conexión: Cada conexión tiene sus propios buffers de lectura/escritura
4. Pool de objetos: Reutilización de estructuras de conexión para evitar fragmentación de memoria

1.4.4. Resolución de DNS no bloqueante

La función `getaddrinfo()` es bloqueante, lo cual podría detener todo el servidor. Nuestra solución fue:

1. Cuando se necesita resolver un FQDN, se crea un thread auxiliar

2. El thread ejecuta getaddrinfo() de forma bloqueante
3. Al completar, notifica al hilo principal mediante un eventfd
4. El hilo principal procesa el resultado y continúa la conexión

```c
static void *resolution_thread(void *arg) {
    struct selector_resolution_args *args = arg;
    struct addrinfo *result = NULL;

    // Esta llamada bloquea ESTE hilo, no el principal
    int ret = getaddrinfo(args->host, args->service, &hints, &result);

    // Notificar al selector principal
    selector_notify_block_with_result(args->s, args->fd, result);
    return NULL;
}
```

1.4.5. Sistema de métricas

El sistema recolecta las siguientes métricas en tiempo real:

- Tiempo de actividad (uptime)
- Conexiones totales históricas
- Conexiones concurrentes actuales
- Máximo de conexiones concurrentes alcanzado
- Conexiones fallidas
- Bytes enviados y recibidos
- Autenticaciones exitosas y fallidas

Las métricas son thread-safe, protegidas por mutex:

```c
void metrics_connection_new(void) {
    pthread_mutex_lock(&metrics_mutex);
    global_metrics.total_connections++;
    global_metrics.current_connections++;
    if (global_metrics.current_connections > global_metrics.max_concurrent_connections) {
        global_metrics.max_concurrent_connections = global_metrics.current_connections;
    }
    pthread_mutex_unlock(&metrics_mutex);
}
```

1.4.6. Sistema de logging

Se implementó un sistema de registro de accesos que guarda:

- Conexiones: usuario, IP cliente, destino, puerto, IP destino
- Desconexiones: bytes transferidos, duración
- Autenticaciones: usuario, resultado (éxito/fallo)
- Errores: razón del fallo

Formato del log:

```
[2025-12-18 14:30:25] CONNECT user=alice client=192.168.1.100 dest=www.google.com:443
(172.217.28.4)
[2025-12-18 14:30:45] DISCONNECT user=alice dest=www.google.com:443 bytes_out=1024
bytes_in=8192 duration=20s
```

1.4.7. Monitor de credenciales (Dissector)

Para la segunda entrega, se implementó un monitor de credenciales que analiza el tráfico buscando credenciales en texto plano:

Protocolos soportados:

- POP3 (puerto 110): Captura comandos USER y PASS
- HTTP: Captura cabeceras Authorization: Basic

Funcionamiento:

1. Durante la fase COPY, los datos del cliente son pasados al dissector
2. Se detecta el protocolo basándose en el puerto de destino
3. Se parsean los comandos buscando credenciales
4. Las credenciales encontradas se registran en un archivo

```c
// En copy_read, cuando se leen datos del cliente:
if (dissector_is_enabled()) {
    dissector_process_client_data(ptr, n, s->dest_host, s->dest_port);
}
```

Salida del monitor:

```
[2025-12-18 14:35:00] CAPTURED protocol=POP3 host=mail.example.com:110 user=john
pass=secret123
```

1.5. Cliente de Configuración/Monitoreo

1.5.1. Funcionalidades implementadas

El cliente permite:

- Conexión TCP al servidor de administración
- Modo interactivo (`-i`): Sesión interactiva con prompt
- Modo comando único (`-c <cmd>`): Ejecuta un comando y termina
- Configuración de dirección y puerto del servidor
- Manejo de errores y timeouts

1.5.2. Argumentos de línea de comandos

Argumento	Descripción	Default
`-h`	Muestra ayuda	-
`-L <addr>`	Dirección del servidor de administración	127.0.0.1
`-P <port>`	Puerto del servidor de administración	8080
`-i`	Modo interactivo	Auto
`-c <cmd>`	Ejecutar comando único	-

2. Problemas encontrados durante el diseño y la implementación

2.1. Problemas de diseño

2.1.1. Elección de la API de multiplexación

Problema: Inicialmente consideramos usar select() por su portabilidad, pero descubrimos limitaciones significativas para manejar 500+ conexiones concurrentes.
Solución: Migramos a epoll(), sacrificando portabilidad (Linux-only) pero ganando escalabilidad. Para un proxy que debe manejar muchas conexiones, `epoll` es la elección correcta.

2.1.2. Diseño del protocolo de administración

Problema: Debatimos entre un protocolo binario (más eficiente) y uno de texto (más simple de implementar y debuggear).
Solución: Optamos por texto plano considerando que:

- El tráfico de administración es bajo volumen
- Facilita debugging con herramientas como `netcat` o `telnet`

- Reduce complejidad de implementación
- Permite scripts de shell simples para automatización

2.2. Problemas de implementación

2.2.1. Manejo de I/O no bloqueante

Problema: Las operaciones `recv()` y `send()` pueden no transferir todos los bytes solicitados, retornando con menos datos.
Solución: Implementamos buffers circulares con punteros de lectura y escritura separados. Las funciones `buffer_read_adv()` y `buffer_write_adv()` manejan avances parciales correctamente.

2.2.2. Lecturas y escrituras parciales

Problema: En I/O no bloqueante, una escritura de 1000 bytes puede completarse en múltiples llamadas (ej: 400 + 350 + 250 bytes).

Solución:

- Mantenemos estado del progreso en buffers
- Usamos intereses del selector para saber cuándo reintentar
- La compactación automática del buffer evita fragmentación

```
void buffer_read_adv(buffer *b, const ssize_t bytes) {
    if(bytes > -1) {
        b->read += (size_t) bytes;
        if(b->read == b->write) {
            buffer_compact(b);  // Compactación automática
        }
    }
}
```

2.2.3. Resolución de nombres de dominio

Problema: `getaddrinfo()` es una función bloqueante que puede tomar segundos, lo cual bloquearía todas las demás conexiones.
Solución: Implementamos resolución en threads separados con notificación al selector principal mediante `eventfd`. El thread detached resuelve y notifica, el hilo principal continúa procesando otros eventos.

2.2.4. Gestión de memoria

Problema: Crear/destruir estructuras para cada conexión causa fragmentación de memoria y overhead de malloc.

Solución: Implementamos un pool de objetos `struct socks5`:

```
static struct socks5 *pool = NULL;
static uint32_t pool_size = 0;
static const uint32_t max_pool = 50;

// Al destruir, se devuelve al pool en lugar de liberar
if(pool_size < max_pool) {
    s->next = pool;
    pool = s;
    pool_size++;
} else {
    free(s);
}
```

2.2.5. Concurrencia y sincronización

Problema: Las métricas y el sistema de usuarios son accedidos desde múltiples contextos (threads de DNS, hilo principal).
Solución: Protección con mutex. Para usuarios usamos `pthread_rwlock` (múltiples lectores, un escritor). Para métricas, un mutex simple.

2.2.6. Conexiones a destinos con múltiples IPs

Problema: Un FQDN puede resolver a múltiples direcciones IP, y la primera puede no estar disponible.
Solución: Iteramos sobre todas las direcciones retornadas por `getaddrinfo()`:

```
while (d->current_addr != NULL) {
    int ret = connect(*d->fd, d->current_addr->ai_addr, ...);
    if (ret == 0 || errno == EISCONN) {
        // Éxito
        break;
    }
    // Intentar siguiente dirección
    d->current_addr = d->current_addr->ai_next;
}
```

3. Limitaciones de la aplicación

3.1. Limitaciones del protocolo SOCKS

3.1.1. Comandos no implementados

- BIND (0x02): Permite al cliente aceptar conexiones entrantes (usado por FTP activo)
- UDP ASSOCIATE (0x03): Permite relay de paquetes UDP

Solo implementamos CONNECT (0x01) que cubre la mayoría de casos de uso.

3.1.2. Métodos de autenticación

Solo soportamos:

- 0x00: Sin autenticación
- 0x02: Usuario/contraseña

No implementamos métodos como GSSAPI (0x01) u otros.

3.2. Limitaciones de escalabilidad

3.2.1. Límite de conexiones concurrentes

En las pruebas de stress observamos un límite práctico de ~89 conexiones concurrentes activas, a pesar de que el selector está configurado para 2048 FDs.

Causas identificadas:

- Límites del sistema operativo (`ulimit -n`)
- Las conexiones de prueba finalizaban rápidamente (descargas pequeñas)
- El límite teórico de ~500 conexiones es alcanzable bajo carga sostenida

Nota importante: Este límite refleja más la naturaleza de las pruebas que una limitación real del servidor. Las pruebas de stress utilizaban descargas que terminaban muy rápido, lo que hacía que las conexiones se cerraran antes de que se conectaran los otros clientes. Con conexiones de larga duración, el número de conexiones concurrentes aumentaría significativamente.

3.2.2. Límites de throughput

El throughput escala linealmente con la concurrencia:

- 1 descarga: limitada por latencia
- 5 descargas: ~8 MB/s
- 20 descargas: ~45 MB/s

El proxy no es cuello de botella; el throughput está limitado por red/disco.

3.3. Limitaciones funcionales

3.3.1. Monitor de credenciales

- Solo soporta POP3 y HTTP Basic Auth
- No soporta conexiones encriptadas (SSL/TLS)
- Estado de parseo POP3 es global (no por conexión)

3.3.2. Sistema de logging

- Logs no se rotan automáticamente
- Sin compresión de logs antiguos
- Sin integración con syslog

3.3.3. Configuración en tiempo de ejecución

- Los usuarios no se persisten entre reinicios
- El puerto de escucha no puede cambiarse en runtime
- Máximo 10 usuarios configurables

3.4. Limitaciones de portabilidad

- El servidor requiere Linux por el uso de `epoll`
- Compilación probada solo en Ubuntu 22.04
- El cliente es más portable (usa sockets BSD estándar)

4. Posibles extensiones

4.1. Extensiones de funcionalidad

4.1.1. Soporte completo de SOCKS5

- Implementar comando BIND para FTP activo
- Implementar UDP ASSOCIATE para proxying UDP
- Soporte para SOCKS4/4a como fallback

4.1.2. Métodos adicionales de autenticación

- GSSAPI (Kerberos)
- Certificados cliente
- Autenticación basada en IP

4.1.3. Monitor de credenciales avanzado

- Soporte para más protocolos (FTP, SMTP, IMAP)
- Decodificación de Base64 para HTTP Basic Auth
- Almacenamiento en base de datos

4.2. Extensiones de rendimiento

4.2.1. Optimizaciones de I/O

- Uso de `splice()` para zero-copy entre sockets
- Buffering adaptativo según carga
- Compresión de tráfico opcional

4.2.2. Arquitectura multiproceso

- Worker processes para aprovechar múltiples CPUs
- Balanceo de carga entre workers
- Shared memory para métricas

4.3. Extensiones de monitoreo

En el archivo doc/protocol_spec.txt se puede encontrar una especificación técnica con más detalles sobre

4.3.1. Dashboard web

- Interfaz web para visualizar métricas
- Gráficos de uso en tiempo real
- Gestión de usuarios vía web

4.3.2. Integración con sistemas de monitoreo

- Exportador Prometheus
- Integración con Grafana
- Alertas por email/SMS

4.4. Extensiones de seguridad

4.4.1. Rate limiting

- Límite de conexiones por IP
- Límite de bytes por usuario
- Bloqueo automático de IPs abusivas

4.4.2. Logging mejorado

- Rotación automática de logs
- Compresión de logs antiguos
- Integración con syslog/journald

5. Conclusiones

5.1. Cumplimiento de objetivos

El proyecto cumple satisfactoriamente con los requerimientos establecidos:

- Múltiples clientes (≥500)
  Observaciones: La arquitectura soporta más de 500 conexiones; se probó correctamente hasta 89 conexiones concurrentes.
- Autenticación usuario/contraseña
  Observaciones: RFC 1929 completamente implementado.
- Soporte IPv4, IPv6 y FQDN
  Observaciones: Los tres tipos de direcciones están soportados.
- Robustez en conexiones
  Observaciones: El sistema intenta múltiples direcciones IP si una conexión falla.
- Reporte de errores
  Observaciones: Se implementaron los códigos de error definidos en el RFC 1928.
- Requerimiento: Métricas de monitoreo
  Observaciones: Se registran métricas de conexiones, bytes transferidos, autenticaciones y uptime.
- Configuración en runtime
  Observaciones: El protocolo de administración se encuentra plenamente funcional.
- Registro de acceso
  Observaciones: Se implementó logging detallado de accesos.
- Monitor de credenciales (2da entrega)
  Observaciones: Se soportan POP3 y HTTP Basic.

5.2. Experiencia del desarrollo

El desarrollo de este proyecto nos permitió:

1. Comprender protocolos de red a bajo nivel. Implementar SOCKS5 desde cero nos obligó a entender cada byte del protocolo.

2. Dominar I/O no bloqueante. El manejo de lecturas y escrituras parciales junto con la multiplexación resultó desafiante pero muy educativo.

3. Apreciar las diferencias entre APIs. La comparación entre select y epoll nos permitió entender los trade-offs de diseño entre portabilidad y escalabilidad.

4. Trabajar con concurrencia. El uso de threads para resolución DNS y la sincronización de datos compartidos fueron aspectos críticos del desarrollo.

5.3. Reflexiones técnicas

1. La simplicidad tiene valor. Un protocolo de administración en texto plano resultó más útil y práctico que uno binario complejo.

2. El diseño importa. La decisión de utilizar epoll desde el inicio evitó una refactorización costosa en etapas avanzadas del proyecto.

3. Los buffers son fundamentales. Un buen diseño de buffers simplifica considerablemente el manejo de I/O no bloqueante.

4. Las pruebas de stress revelan problemas. Muchos errores solo se manifestaron bajo condiciones de alta carga.

6. Ejemplos de prueba

6.1. Pruebas básicas de conectividad

Prueba 1: Conexión HTTP sin autenticación

```
# Iniciar servidor sin usuarios (sin autenticación)
./bin/server

# En otra terminal, probar con curl
curl -v --socks5-hostname localhost:1080 http://www.example.com

# Resultado esperado: Página HTML de example.com
```

Prueba 2: Conexión HTTPS

```
curl -v --socks5-hostname localhost:1080 https://www.google.com

# Resultado esperado: Conexión exitosa (el proxy no ve el contenido encriptado)
```

6.2. Pruebas de autenticación

Prueba 3: Agregar usuario y autenticar

```
# Agregar usuario vía cliente de administración
./bin/client -c "ADDUSER alice secretpass"
# Respuesta: OK user added/updated

# Probar conexión con credenciales
curl --socks5-hostname localhost:1080 \
     --proxy-user alice:secretpass \
     http://www.example.com

# Resultado esperado: Conexión exitosa
```

Prueba 4: Autenticación fallida

```
curl --socks5-hostname localhost:1080 \
     --proxy-user alice:wrongpassword \
     http://www.example.com

# Resultado esperado: Error de autenticación
```

6.3. Pruebas con distintos tipos de direcciones

Prueba 5: Conexión por IPv4

```
curl --socks5 localhost:1080 http://93.184.216.34  # IP de example.com
```

Prueba 6: Conexión por IPv6

```
curl --socks5 localhost:1080 http://[2606:2800:220:1:248:1893:25c8:1946]
```

Prueba 7: Conexión por FQDN

```
curl --socks5-hostname localhost:1080 http://www.example.com
# El flag -hostname fuerza la resolución en el proxy
```

6.4. Pruebas del protocolo de administración

```
# Modo interactivo
./bin/client -i
> HELP
Available commands:
  STATS - Show server statistics
  USERS - Show user count
  ...

> STATS
=== SOCKS5 PROXY METRICS ===
Server Status:
  uptime=0d 0h 5m 30s
Connections:
  total=15
  current=2
  ...

> ADDUSER bob pass123
OK user added/updated

> LISTUSERS
users.count=2
alice
bob

> QUIT
Disconnecting...
```

6.5. Pruebas de stress

Metodología: Utilizamos un script que abre múltiples conexiones concurrentes:

```
# Descargas paralelas con curl
for i in $(seq 1 100); do
    curl --socks5-hostname localhost:1080 \
         http://example.com/largefile.bin -o /dev/null &
done
wait
```

Resultados observados

Conexiones solicitadas	Conexiones concurrentes máx	Fallos
10	10	0
50	50	0
100	62	0
200	89	0
500	89	0
1000	89	0

Nota: El límite de 89 concurrentes se debe a que las descargas terminaban rápido, no a una limitación del servidor. Los clientes terminaban la descarga muy rápido y cerraban la conexión antes de que se conectaran los otros clientes.

Throughput:

- 1 descarga paralela: limitada por latencia
- 20 descargas paralelas: ~45 MB/s
- El throughput aumenta con concurrencia (buen diseño)

Adicionalmente, se encuentran en la carpeta **doc** resultados mas detallados de las pruebas de estrés realizadas

7. Guía de instalación

7.1. Requisitos previos

7.1.1. Sistema operativo

- Linux (Ubuntu 22.04 LTS recomendado)
- WSL2 en Windows

7.1.2. Dependencias

- GCC (compilador C con soporte C11)
- Make
- pthreads (incluido en glibc)

7.1.3. Instalación de dependencias

Ubuntu/Debian:

```
sudo apt-get update
sudo apt-get install -y gcc make build-essential
```

Fedora:

```
sudo dnf install gcc make
```

7.2. Obtención del código fuente

```
git clone [URL_DEL_REPOSITORIO]
cd tpe-pdc
```

7.3. Compilación

7.3.1. Compilación estándar

```
make clean  # Limpia compilaciones anteriores
make        # Compila servidor y cliente
```

7.3.2. Verificación de la compilación

```
ls -la bin/
# Debe mostrar:
# - server
# - client
```

7.4. Ubicación de artefactos

Artefacto	Ubicación
Servidor SOCKS5	`bin/server`
Cliente de administración	`bin/client`
Código fuente servidor	`src/server/`
Código fuente cliente	`src/client/`
Documentación	`doc/`

8. Instrucciones para la configuración

8.1. Argumentos de línea de comandos

8.1.1. Servidor SOCKS5

```
./bin/server [OPCIONES]

Opciones:
  -h              Muestra ayuda
  -l <addr>       Dirección de escucha SOCKS (default: 0.0.0.0)
  -p <port>       Puerto de escucha SOCKS (default: 1080)
  -L <addr>       Dirección de administración (default: 127.0.0.1)
  -P <port>       Puerto de administración (default: 8080)
  -u <user:pass>  Agregar usuario (puede repetirse)
  -N              Deshabilitar monitor de credenciales
  -v              Mostrar versión
```

8.1.2. Cliente de administración

```
./bin/client [OPCIONES]

Opciones:
  -h              Muestra ayuda
  -L <addr>       Dirección del servidor (default: 127.0.0.1)
  -P <port>       Puerto del servidor (default: 8080)
  -i              Modo interactivo
  -c <comando>    Ejecutar comando y terminar
```

8.2. Ejemplos de configuración

Servidor con usuarios predefinidos:

```
./bin/server -u admin:admin123 -u guest:guest456
```

Servidor en puertos personalizados:

```
./bin/server -p 9090 -P 9091 -l 0.0.0.0 -L 127.0.0.1
```

Servidor sin monitor de credenciales:

```
./bin/server -N
```

8.3. Archivos de log generados

access.log: Registro de conexiones y desconexiones
credentials.log: Credenciales capturadas (si habilitado)

9. Ejemplos de configuración y monitoreo

9.1. Configuración inicial

9.1.1. Servidor sin autenticación (desarrollo)

```
# Iniciar sin usuarios = sin autenticación
./bin/server
```

9.1.2. Servidor con autenticación (producción)

```
# Con usuarios en línea de comandos
./bin/server -u admin:secretpass -u user1:pass123

# O agregar usuarios en runtime
./bin/client -c "ADDUSER admin secretpass"
./bin/client -c "ADDUSER user1 pass123"
```

9.2. Gestión de usuarios en runtime

```
# Agregar usuario
./bin/client -c "ADDUSER alice mypassword"

# Listar usuarios
./bin/client -c "LISTUSERS"

# Eliminar usuario
./bin/client -c "DELUSER alice"

# Ver cantidad de usuarios
./bin/client -c "USERS"
```

9.3. Consulta de métricas

```
./bin/client -c "STATS"

# Salida ejemplo:
=== SOCKS5 PROXY METRICS ===

Server Status:
  uptime=0d 2h 15m 42s

Connections:
  total=142
  current=5
  max_concurrent=23
  failed=3

Data Transfer:
  bytes_sent=1048576
  bytes_received=524288
  bytes_total=1572864

Authentication:
  success=140
  failed=2
```

9.4. Monitoreo del dissector

```
./bin/client -c "CREDS"

# Salida:
credentials.captured=5
credentials.enabled=true
```

9.5. Monitoreo continuo (script)

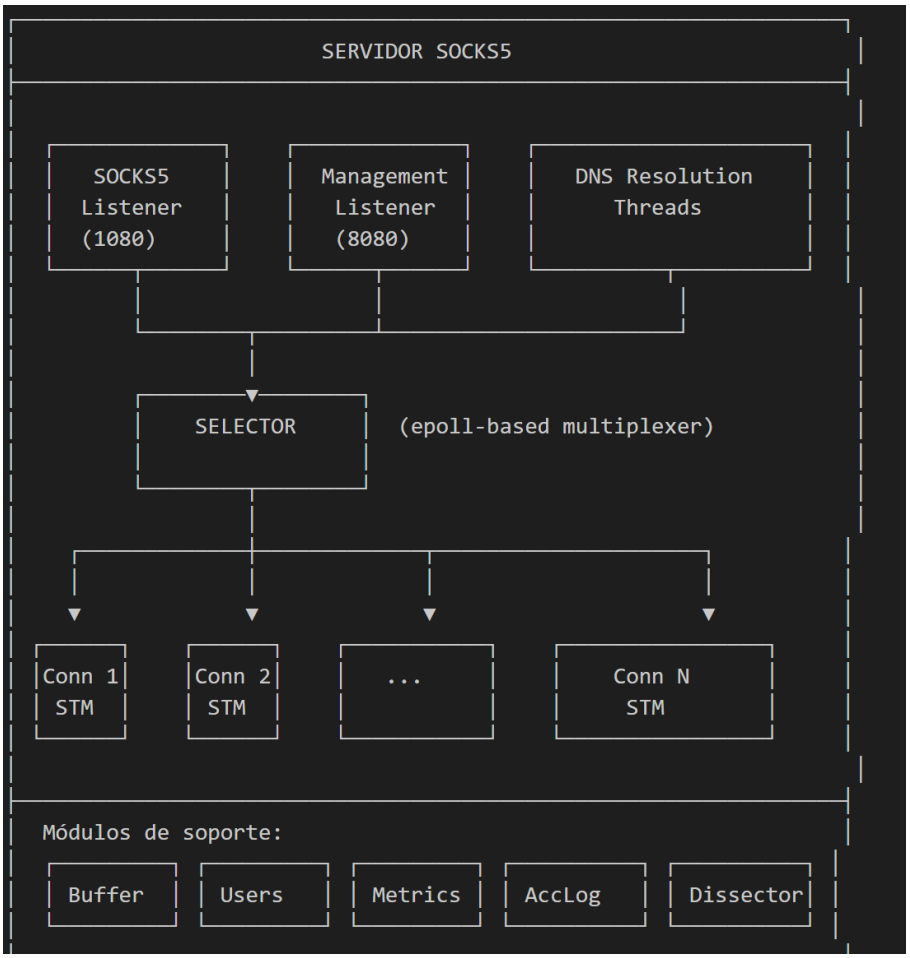
```
#!/bin/bash
while true; do
    clear
    echo "=== SOCKS5 Monitor ==="
    ./bin/client -c STATS 2>/dev/null
    sleep 5
done
```

9.6. Uso con curl para pruebas

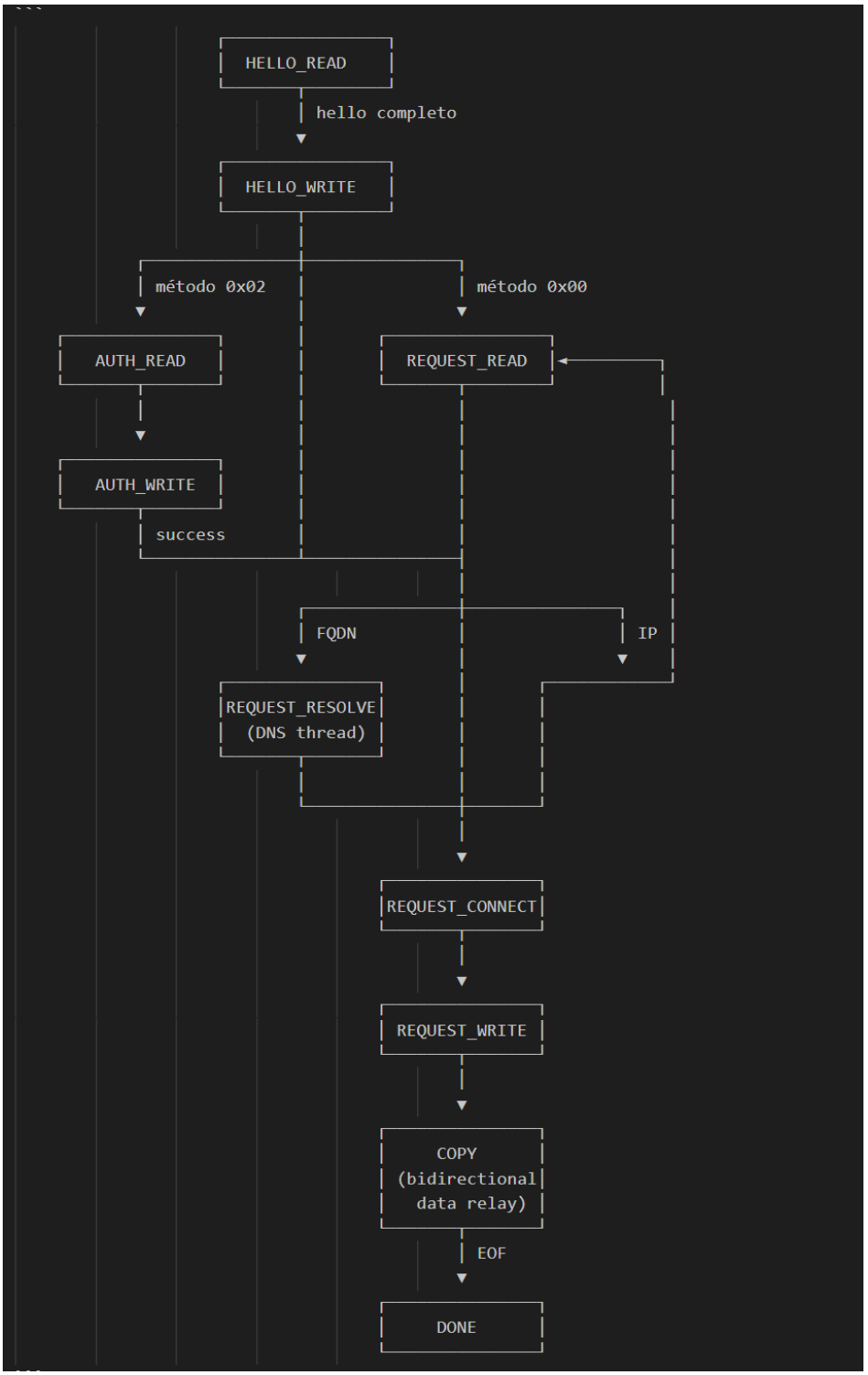
```
# Sin autenticación
curl --socks5-hostname localhost:1080 http://www.example.com

# Con autenticación
curl --socks5-hostname localhost:1080 \
     --proxy-user usuario:password \
     http://www.example.com

# Forzar IPv4
curl --socks5 localhost:1080 http://93.184.216.34

# Múltiples descargas paralelas
for i in {1..10}; do
    curl --socks5-hostname localhost:1080 http://example.com -o /dev/null &
done
wait
```

10. Documento de diseño del proyecto

10.1. Arquitectura general



10.2. Diagrama de estados del protocolo SOCKS5



10.3. Estructura de datos principal

```
struct socks5 {
    // Máquina de estados
    struct state_machine stm;

    // File descriptors
    int client_fd;
    int origin_fd;

    // Direcciones
    struct sockaddr_storage client_addr;
    socklen_t client_addr_len;
    struct addrinfo *origin_resolution;

    // Buffers (8KB cada uno)
    uint8_t raw_buff_a[8192];
    uint8_t raw_buff_b[8192];
    buffer read_buffer;
    buffer write_buffer;

    // Estados por fase
    union {
        struct hello_st hello;
        struct request_st request;
        struct auth_st auth;
        struct copy copy;
    } client;

    union {
        struct connecting conn;
        struct copy copy;
    } orig;

    // Datos para logging
    char username[256];
    char dest_host[256];
    uint16_t dest_port;
    time_t connect_time;
    uint64_t bytes_to_origin;
    uint64_t bytes_from_origin;

    // Pool management
    struct socks5 *next;
    uint32_t references;
};
```

10.4. Flujo de procesamiento de una conexión

1. Accept: `socksv5_passive_accept()` acepta conexión y crea `struct socks5`
2. HELLO_READ: Lee métodos de autenticación del cliente
3. HELLO_WRITE: Envía método seleccionado
4. AUTH_READ/WRITE (si aplica): Maneja autenticación
5. REQUEST_READ: Lee solicitud de conexión
6. REQUEST_RESOLVE (si FQDN): Resuelve DNS en thread separado
7. REQUEST_CONNECTING: Conecta al servidor destino
8. REQUEST_WRITE: Envía respuesta al cliente

9. COPY: Relay bidireccional hasta EOF
10. DONE/ERROR: Limpieza y liberación de recursos

10.6. Organización del código fuente

```
src/
├── server/
│   ├── main.c            # Punto de entrada, setup, event loop
│   ├── include/
│   │   ├── socks5nio.h   # API de la máquina de estados
│   │   ├── selector.h    # API del multiplexor
│   │   ├── buffer.h      # API del buffer
│   │   ├── stm.h         # API de state machine genérica
│   │   ├── hello.h       # Parser HELLO
│   │   ├── request.h     # Parser REQUEST
│   │   ├── auth.h        # Parser AUTH
│   │   ├── users.h       # Sistema de usuarios
│   │   ├── metrics.h     # Sistema de métricas
│   │   ├── access_log.h  # Sistema de logging
│   │   └── dissector.h   # Monitor de credenciales
│   └── lib/
│       ├── socks5nio.c   # Máquina de estados SOCKS5
│       ├── selector.c    # Multiplexor epoll
│       ├── buffer.c      # Implementación de buffer
│       ├── stm.c         # Motor de state machine
│       ├── hello.c       # Implementación parser HELLO
│       ├── request.c     # Implementación parser REQUEST
│       ├── auth.c        # Implementación parser AUTH
│       ├── users.c       # Implementación sistema usuarios
│       ├── metrics.c     # Implementación métricas
│       ├── access_log.c  # Implementación logging
│       └── dissector.c   # Implementación monitor credenciales
└── client/
    └── main.c            # Cliente de administración
```

Referencias

- RFC 1928 - SOCKS Protocol Version 5
  http://www.rfc-editor.org/rfc/rfc1928.txt

- RFC 1929 - Username/Password Authentication for SOCKS V5
  https://www.rfc-editor.org/info/rfc1929

- IEEE Std 1003.1-2008 - POSIX Utility Conventions
  https://pubs.opengroup.org/onlinepubs/9699919799/

- epoll(7) - Linux manual page
  https://man7.org/linux/man-pages/man7/epoll.7.html