TRABAJO PRÁCTICO ESPECIAL 2025/1

SERVIDOR PROXY SOCKS5 CON ADMINISTRACIÓN

72.07 PROTOCOLOS DE COMUNICACIÓN

INSTITUTO TECNOLÓGICO DE BUENOS AIRES

Grupo 5

Manuel Ahumada - 64677

Josefina Gonzalez Cornet - 64550

Juan Ignacio Lategana - 64476

Nicolás Priotto - 64663

ÍNDICE

1. DESCRIPCIÓN DE LOS PROTOCOLOS Y APLICACIONES DESARROLLADAS

1.1 SERVIDOR PROXY SOCKS5

El siguiente proyecto implementa un servidor proxy completo basado en SOCKSv5 (RFC 1928) con soporte para autenticación usuario/contraseña (RFC 1929). El servidor maneja conexiones TCP hacia destinos IPv4, IPv6 y nombres de dominio, incorporando resolución DNS asíncrona para evitar bloqueos.

La implementación sigue las especificaciones del RFC comenzando con negociación de autenticación, intercambio de credenciales, procesamiento de solicitudes de conexión, y finalmente transfiriendo datos de forma bidireccional al cliente y servidor destino.

El sistema incluye logging de accesos en formato ISO-8601, registrando usuario, direcciones IP, puertos y resultados para facilitar un posterior análisis.

1.2 PROTOCOLO DE ADMINISTRACIÓN

Se desarrolló un protocolo binario con TCP para administración remota sin reinicio del servidor. Permite gestión completa de usuarios, consulta de métricas en tiempo real y modificación de ciertos parámetros de configuración. La estructura de mensajes utiliza byte de versión, identificador de comando, campo de longitud opcional y payload de datos. Las respuestas incluyen códigos de estado para determinar el resultado de cada operación. La autenticación utiliza un token predefinido que debe proporcionarse al establecer la conexión.

Los comandos soportados incluyen:

- **help**: Mostrar ayuda
- **list-users**: Listar usuarios activos
- **add <user> <pass>**: Agregar usuario
- **del <user>**: Eliminar usuario
- **metrics**: Mostrar estadísticas completas
- **set-log <level>**: Cambiar nivel de logging
- **set-max <num>**: Cambiar máximo de conexiones
- **clear**: Limpiar pantalla
- **quit**: Salir

1.3 SISTEMA DE MÉTRICAS

El sistema recolecta estadísticas organizadas en categorías: conexiones (total histórico, actuales, exitosas, fallidas), transferencia de datos (bytes en ambas direcciones, throughput), autenticación (intentos por método, usuarios únicos), errores categorizados y rendimiento (tiempo promedio de conexión).

Las métricas se actualizan en tiempo real y son accesibles mediante el protocolo de administración para monitoreo continuo del servidor.

1.4 CLIENTE DE ADMINISTRACIÓN

La aplicación cliente proporciona una interfaz interactiva que abstrae la complejidad del protocolo binario. Incluye comandos de alto nivel para visualización de métricas, gestión de usuarios y configuración dinámica. De esta manera se posibilita una mejor experiencia para los usuarios de la aplicación cliente sin implicar caídas del rendimiento del mismo. A pesar de que maneja strings para los comandos, estos se parsean una sola vez e internamente se manejan de forma binaria (códigos de comando para cada uno), por lo que la conexión entre el administrador y el servidor es prácticamente instantánea, como se esperaría en un protocolo binario de tal simpleza.

La autenticación se maneja internamente de forma automática, por lo que no es necesario que los usuarios de la aplicación provean credenciales.

2. PROBLEMAS ENCONTRADOS DURANTE EL DISEÑO Y LA IMPLEMENTACIÓN

2.1 CONCURRENCIA Y SINCRONIZACIÓN

Durante el desarrollo del proyecto, uno de los primeros desafíos fue lograr manejar de forma eficiente múltiples conexiones simultáneas sin que se produjeran bloqueos. Para ello, se implementó un sistema no bloqueante utilizando la función select(), permitiendo utilizar sockets para lograr concurrencia.

Otro inconveniente fue la resolución de nombres de dominio (DNS), ya que sus llamadas eran bloqueantes, y podrían afectar al hilo principal de ejecución. Para resolver este problema, se diseñó un sistema de resolución de dominios en hilos separados, con una notificación asincrónica para informar al hilo principal que la resolución fue completada.

También se identificaron problemas al querer acceder a estructuras compartidas, como aquellas que almacenan las métricas globales. Para solucionarlo, se optó por utilizar mutex para proteger las zonas críticas.

2.2 PARSING DE PROTOCOLOS

Al querer procesar los datos del protocolo SOCKS5, un problema importante fue que éstos podían llegar en fragmentados. Por ello se utilizó una máquina de estados explícita, respaldada por buffers que permiten acumular datos hasta que el mensaje esté completo.

Luego, en el caso específico de la autenticación por usuario y contraseña, se desarrolló un parser híbrido que combina la máquina de estados con el procesamiento detallado byte a byte.

2.3 MANEJO DE MEMORIA

Durante las primeras etapas de testeo, se detectaron pérdidas de memoria y problemas de lectura luego de liberar un dato, que ocurrían cuando las conexiones eran abortadas de forma inesperada. Para solucionar el problema, se implementó el uso de flags que indican cuándo una estructura debía ser destruida y liberada de forma segura.

Por otro lado, se realizaron pruebas para determinar el tamaño de buffer ideal para este proyecto. Se llegó a la conclusión que el tamaño de 64KB era adecuado para generar un buen equilibrio entre eficiencia y simplicidad. Este valor se puede modificar en el archivo *buffer.c*, redefiniendo BUFFER_SIZE al tamaño deseado.

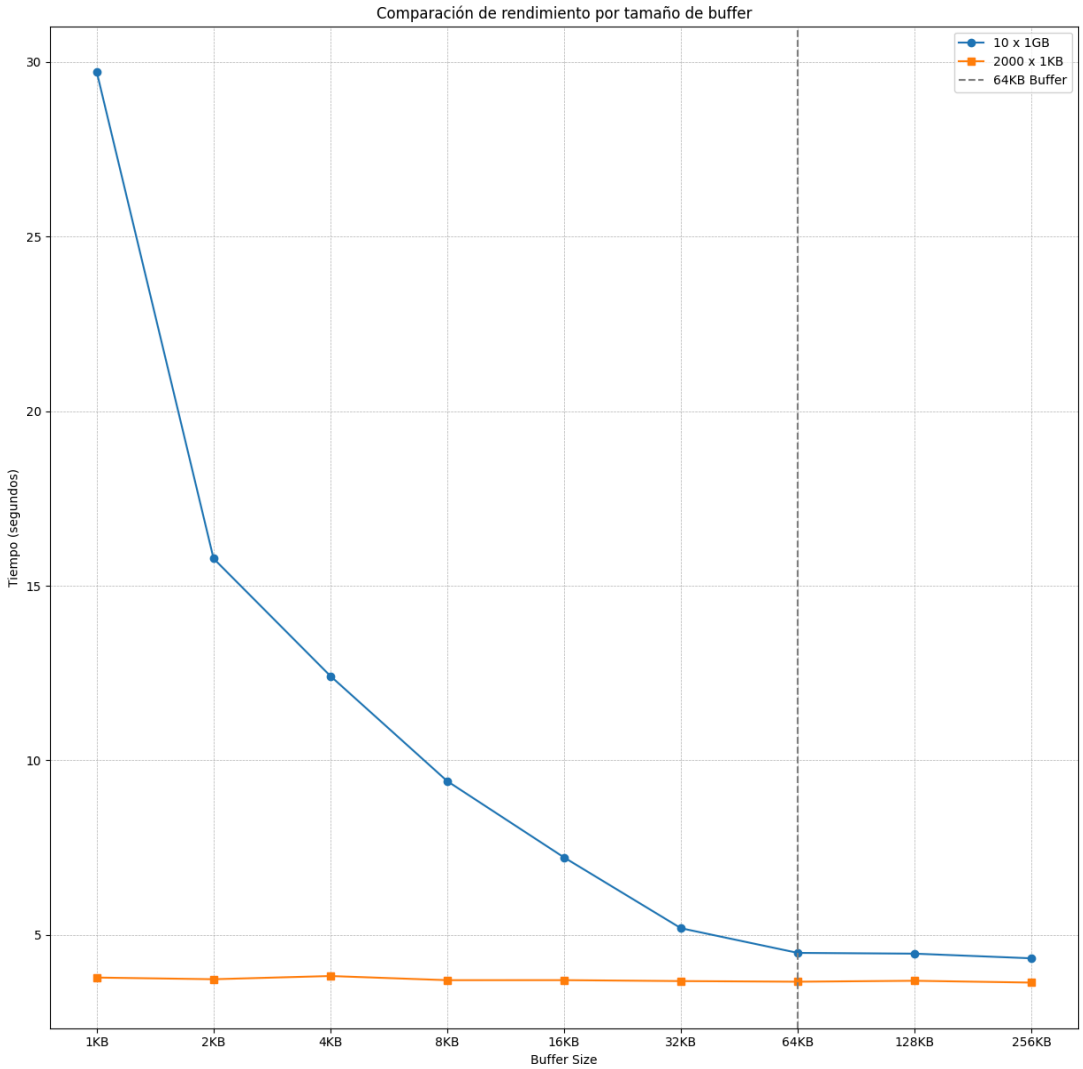

Figura 1. Gráfico de comparación de rendimiento por tamaño de buffer

En este gráfico se pueden ver los resultados de rendimiento de dos tests independientes, uno que consiste de 10 requests de 1GB cada una, y otro de 2000 requests de 1KB cada una. El tiempo que cada uno de ellos tomó fue medido y puesto en este gráfico para formar ambas curvas. Se puede observar que los tiempos para el caso de transferencias grandes de datos es drásticamente menor al agrandar el buffer. En base a esto, se eligió el tamaño de 64KB ya que a partir de dicho tamaño el aumento de performance no es tan notorio, y además hace que para requests de tamaño muy pequeño no se malgaste espacio.

2.4 ROBUSTEZ DE RED

No siempre es posible conectarse a la primera dirección IP asociada a un nombre de dominio. Es por ello que se implementó una lógica que itera por todas las direcciones IP devueltas por la resolución DNS, hasta encontrar una que funcione.

Finalmente, se consideraron aquellas situaciones donde el cliente o el servidor remoto podían cerrar sus conexiones de forma abrupta. Para ello, se incluyó una verificación explícita del estado de los sockets, para evitar posibles problemas de memoria.

3. LIMITACIONES DE LA APLICACIÓN

3.1 LIMITACIONES FUNCIONALES

A nivel funcional, la aplicación presenta ciertas restricciones que responden tanto a decisiones de diseño como a las prioridades de implementación.

En primer lugar, el servidor admite por defecto hasta 500 conexiones concurrentes. Este límite fue establecido con el objetivo de evitar la necesidad de modificar configuraciones del sistema operativo en los entornos donde se ejecute el proyecto. No obstante, el valor es configurable y puede incrementarse, aunque hacerlo podría requerir ajustes adicionales en el sistema. Es importante tener en cuenta que si por ejemplo se establece un máximo de 20 conexiones, todas las conexiones concurrentes que lleguen cuando ya están ocupados los 20 espacios, van a ser rechazadas por el servidor socks5d.

Por otra parte, la autenticación implementada carece de cifrado en el canal de comunicación, lo que representa una debilidad importante en contextos donde se requiere confidencialidad, como redes inseguras o entornos corporativos con requisitos de seguridad más estrictos.

3.2 LIMITACIONES DE RENDIMIENTO

En términos de rendimiento, la función select() impone una barrera de escalabilidad, especialmente en sistemas Linux dónde se encuentran herramientas más eficientes como el uso de “epoll”.

Los buffers de tamaño fijo, establecidos en 64KB, suelen funcionar para la mayoría de las transferencias, pero pueden verse ineficientes con flujos de datos tanto muy chicos, como muy grandes, donde sería preferible un enfoque dinámico.

Además, la resolución DNS se realiza mediante la creación de un nuevo hilo por cada una. Esto podría escalar mal bajo una carga alta de solicitudes.

3.3 LIMITACIONES DE SEGURIDAD

Respecto a la seguridad del protocolo, existen varias áreas que requieren atención. El token usado para acceder a la interfaz de administración se encuentra hardcodeado en el código fuente, lo que representa un riesgo si el binario o el repositorio llegaran a estar comprometidos.

No se implementan mecanismos para limitar la cantidad de intentos de autenticación fallidos, lo que abre la posibilidad a ataques de fuerza bruta.

4. POSIBLES EXTENSIONES

4.1 FUNCIONALIDADES ADICIONALES

Se podría extender el soporte del protocolo incorporando un balanceador de carga para múltiples destinos, un caché para resolver solicitudes DNS, entre otros.

Se puede realizar una cola de espera para las conexiones que llegan concurrentemente al servidor cuando ya el máximo de conexiones concurrentes fue alcanzado, de modo que en vez de ser rechazadas por el servidor, son procesadas con un tiempo de espera.

4.2 MEJORAS DE RENDIMIENTO

En términos de rendimiento, el paso siguiente sería migrar de select a epoll para obtener una mayor escalabilidad. También sería acorde implementar un pool de threads exclusivamente para resolución de DNS, y tamaños de buffers dinámicos.

4.3 ADMINISTRACIÓN Y MONITOREO

Se podría implementar una interfaz de tipo web para la administración, junto con un soporte de alertas automáticas, respaldo y generación de logs más específicos.

5. CONCLUSIONES

El desarrollo del proyecto resultó en una implementación robusta que cumple todos los objetivos funcionales y no funcionales especificados en el enunciado. La arquitectura modular facilita el mantenimiento y extensiones futuras, mientras que la separación de responsabilidades permite modificaciones independientes.

El sistema de métricas proporciona visibilidad operacional completa. El protocolo de administración binario demuestra eficiencia y extensibilidad apropiadas. El manejo de concurrencia mediante I/O no bloqueante y threads para operaciones bloqueantes evita problemas de rendimiento comunes.

El proceso proporcionó experiencia valiosa en diseño de sistemas distribuidos, debugging de aplicaciones de red y optimización de rendimiento, contribuyendo al entendimiento de protocolos de comunicación, la programación con sockets, y su implementación práctica.

6. EJEMPLOS DE PRUEBA

6.1 PRUEBAS BÁSICAS DEL SERVIDOR SOCKS5

Las pruebas iniciales se basaron en la realización de un curl simple para verificar funcionalidad. Comandos como el siguiente demostraron manejo correcto de autenticación y establecimiento de conexiones.

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

La resolución DNS se probó usando nombres de dominio, IPv4 e IPv6 y resultó ser exitoso.

6.2 PRUEBAS DEL PROTOCOLO DE ADMINISTRACIÓN

Se realizó la ejecución y conexión al servidor cliente y se ejecutaron todos los comandos disponibles en una serie de momentos dados para garantizar su correcto funcionamiento.

6.3 PRUEBAS DE CARGA Y RENDIMIENTO

Se desarrollaron los siguientes programas: *server.py* y *server2.py*, que montan un servidor HTTP que devuelve 1KB o 1GB de datos correspondientemente. A su vez su utilizó test.sh para llevar a cabo múltiples conexiones concurrentes a esos servidores.

server.py

```
#!/usr/bin/env python3
# save as server.py
from http.server import BaseHTTPRequestHandler, HTTPServer

class SimpleHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        data = b"A" * 1024  # 1 KB of dummy data
        self.send_response(200)
        self.send_header('Content-type', 'application/octet-stream')
        self.send_header('Content-Length', str(len(data)))
        self.end_headers()
        self.wfile.write(data)

    def log_message(self, format, *args):
        pass  # Silence the logs

PORT = 3000
print(f"Serving 1KB responses on http://localhost:{PORT}")
```

```
    HTTPServer(('', PORT), SimpleHandler).serve_forever()
```

server2.py

```
#!/usr/bin/env python3
from http.server import BaseHTTPRequestHandler, HTTPServer
from socketserver import ThreadingMixIn

CHUNK_SIZE = 1024 * 1024  # 1 MB
NUM_CHUNKS = 1024         # 1024 MB = 1 GB

class ThreadedHTTPServer(ThreadingMixIn, HTTPServer):
    """Handle requests in separate threads."""

class OneGBHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        total_size = CHUNK_SIZE * NUM_CHUNKS
        self.send_response(200)
        self.send_header("Content-Type", "application/octet-stream")
        self.send_header("Content-Length", str(total_size))
        self.end_headers()

        data = b"A" * CHUNK_SIZE
        for _ in range(NUM_CHUNKS):
            try:
                self.wfile.write(data)
            except BrokenPipeError:
                break  # client disconnected

    def log_message(self, format, *args):
        return  # Silence logs

if __name__ == "__main__":
    server_address = ('', 3000)
    httpd = ThreadedHTTPServer(server_address, OneGBHandler)
    print("Serving 1GB per request on http://localhost:3000")
     httpd.serve_forever()
```

test.sh

```
#!/bin/bash

CONCURRENT=20
PROXY="socks5h://localhost:1080"
USER="charlie:chocolate"
URL="http://127.0.0.1:3000"
TMP=$(mktemp)
touch "$TMP"

fetch() {
  local id=$1
   if curl --silent --output /dev/null --proxy "$PROXY" --proxy-user "$USER"
"$URL"; then
    echo "[$id] Success"
    echo "success" >> "$TMP"
  else
    echo "[$id] Failed"
    echo "fail" >> "$TMP"
  fi
}

export -f fetch
export PROXY USER URL TMP

seq $CONCURRENT | xargs -n1 -P$CONCURRENT -I{} bash -c 'fetch "$@"' _ {}

# Count results
SUCCESSES=$(grep -c success "$TMP")
FAILURES=$(grep -c fail "$TMP")
rm "$TMP"

echo ""
echo "✅ Successes: $SUCCESSES"
echo "❌ Failures:  $FAILURES"
```

Para los testeos, en una terminal se ejecutó el servidor redirigiendo la salida de error a un archivo output.txt para facilitar el debuggeo (`make clean && make && ./bin/socks5d > out.txt 2>&1`). En otra terminal se levantó el servidor deseado de python (`python3 server.py`). En una tercera terminal se ejecutó `test.sh` que imprime en salida estándar la cantidad de éxitos y fallos que hubo durante la ejecución.

Se testeó con una variedad de órdenes de concurrencia. Se decidió testear con un servidor en localhost para que sea más rápido y así poder ejecutar más pruebas en un tiempo dado. Sin embargo, también se realizaron tests con conexiones a servidores pesados de internet, como por ejemplo editando la URL en test.sh y colocando alguna de las siguientes en su lugar: http://nbg1-speed.hetzner.com/100MB.bin, http://nbg1-speed.hetzner.com/1MB.bin, http://nbg1-speed.hetzner.com/1GB.bin.

A su vez se monitorearon las métricas durante todas las pruebas para verificar la escalabilidad.

6.4 PRUEBAS DE ROBUSTEZ

Las pruebas incluyeron conexiones sin autenticación o con credenciales de autenticación inválida, conexiones a servidores no disponibles, resoluciones DNS de dominios inexistentes, entre otros. Se verificó manejo apropiado de códigos de error SOCKS5 y recuperación limpia de fallos. A continuación se presentan algunos casos a modo de ejemplo:

Caso 1: conexión sin autenticación

```
$ curl --proxy socks5h://localhost:1080
https://nbg1-speed.hetzner.com/1MB.bin > /dev/null

curl: (7) No authentication method was acceptable. (It is quite likely that
the SOCKS5 server wanted a username/password, since none was supplied to the
server on this connection.)
```

Caso 2: conexión con autenticación inválida

```
$ curl --proxy socks5h://localhost:1080 --proxy-user charlie:dulcedeleche
https://nbg1-speed.hetzner.com/1MB.bin > /dev/null

curl: (7) User was rejected by the SOCKS5 server (1 1).
```

Caso 3: conexión a un servidor no disponible

```
$ curl --proxy socks5h://localhost:1080 --proxy-pass charlie:chocolate
https://skibidi.toilet.world.game/ > /dev/null

curl: (7) Can't complete SOCKS5 connection to skibidi.toilet.world.game:443.
(4)
```

7. GUÍA DE INSTALACIÓN

7.1 REQUISITOS DEL SISTEMA

- Sistema Linux
- Compilador C compatible con C11
- Make

7.2 PROCESO DE COMPILACIÓN

Extraer código fuente y ejecutar:

```
make clean
make all
```

Se generan binarios ejecutables en /bin/: `socks5d` (servidor) y `client` (administración).

8. INSTRUCCIONES PARA LA CONFIGURACIÓN

8.1 CONFIGURACIÓN DEL SERVIDOR

Parámetros principales:

- **-l** dirección: IP de escucha SOCKS5 (default: 0.0.0.0)
- **-p** puerto: Puerto SOCKS5 (default: 1080)
- **-L** dirección: IP administración (default: 127.0.0.1)
- **-P** puerto: Puerto administración (default: 8080)
- **-u** usuario:contraseña: Agregar usuario (hasta 10 veces)
- **-a <archivo>**: Archivo de logs de acceso
- **-h**: Mostrar ayuda
- **-v**: Mostrar versión

8.2 ARCHIVO DE USUARIOS

Se añadió la posibilidad de proveer un archivo *users.csv* en la raíz del proyecto con usuarios precargados para la conexión al servidor socks5. Este archivo debe tener formato CSV con punto y coma como delimitador. Por ejemplo:

```
alice;wonderland
bob;builder
charlie;chocolate
```

Este archivo se lee al inicio de la ejecución del servidor. Los usuarios que el cliente administrador agregue con el parámetro **-u <user> <pass>** no se agregan al archivo (no son persistentes). Sin la presencia de *users.csv*, el servidor comenzará sin una lista de usuarios disponible y será estrictamente necesario que se conecté un administrador y realice la configuración necesaria.

8.3 CONFIGURACIÓN DE RED

A la hora de configurar la red del servidor, la dirección IP en la que escucha el servicio puede ajustarse según el alcance deseado:

La opción -l se utiliza para indicar en qué dirección IP debe escuchar la aplicación o servidor. Por ejemplo, `-l 0.0.0.0` hace que escuche en todas las interfaces IPv4 disponibles, mientras que `-l ::` permite escuchar tanto en IPv6 como en IPv4 si el sistema lo permite. Si se usa `-l 127.0.0.1`, el servicio quedará limitado al acceso local, es decir, solo podrá ser usado desde la misma máquina. En cambio, al especificar una IP concreta como `-l 192.168.1.100`, se restringe el servicio a esa interfaz en particular, útil cuando se quiere controlar exactamente por dónde se reciben conexiones.

8.4 LOGGING

Hay 4 niveles disponibles: 0=DEBUG, 1=INFO, 2=ERROR, 3=FATAL. El valor por defecto es el de INFO. A través de la aplicación cliente de administración se puede modificar dinámicamente el nivel de logging con el comando `set-log <level>`.

El logging se realiza por salida estándar. Se puede redirigir esta salida con el uso del parámetro opcional **-a <archivo>** al iniciar el servidor.

9. EJEMPLOS DE CONFIGURACIÓN Y MONITOREO

9.1 CONFIGURACIÓN

Ejecución del servidor.

```
./bin/socks5d
```

De esta manera se ejecuta con los valores defecto, que serían:

- Listen en 0.0.0.0:1080 para el servidor socks5
- Listen en 127.0.0.1:8080 para la aplicación cliente
- Logging a nivel INFO
- Usuarios precargados de *users.csv* (solo si existiese este archivo en la raíz del proyecto)

9.2 MONITOREO CON CLIENTE

Conexión al servidor de administración:

```
./bin/client 127.0.0.1 8080
```

Se abre una interfaz donde se pueden llamar los comandos explicados anteriormente (en la sección 1.2).

9.3 ANÁLISIS DE LOGS

Los logs son claros y brindan la información necesaria para comprender los movimientos realizados por el programa en tiempo de ejecución. Los logs de acceso tienen el siguiente formato (separado por tabs):

```
fecha usuario tipo IP_origen puerto_origen destino puerto_destino status_code
```

Los status code son los propios del protocolo socks5 de acuerdo al RFC 1928.

Para facilitar el análisis de estos logs de acceso en volúmenes muy grandes de datos, se pueden ejecutar por ejemplo los siguientes comandos estándares:

```
# análisis de conexiones por usuario (imprime cantidad y nombre de usuario)
awk -F'\t' '{print $2}' access.log | sort | uniq -c

# análisis de destinos más populares (imprime cantidad y destino)
awk -F'\t' '{print $6}' access.log | sort | uniq -c | sort -nr

# análisis de conexiones por hora (imprime cantidad y hora)
awk -F'\t' '{print substr($1,12,2)}' access.log | sort | uniq -c
```

10. DOCUMENTO DE DISEÑO DEL PROYECTO

10.1 ARQUITECTURA GENERAL

El sistema utiliza una arquitectura modular con separación clara de responsabilidades. El servidor principal maneja tanto conexiones SOCKS5 como administración simultáneamente usando un selector I/O no bloqueante que multiplexa todas las conexiones en un hilo único.

Los componentes están organizados en:

- Servidor principal (*main.c*)
- Manejo de conexiones SOCKS5 (*connection.c*, *socks5.c*)
- Servidor de administración (*admin_protocol.c*)
- Componentes compartidos (usuarios, métricas, logging, utilidades)

10.2 FLUJO DE PROCESAMIENTO SOCKS5 - MÁQUINA DE ESTADOS

El flujo de procesamiento del protocolo socks5 se modela a través de una máquina de estados (que puede considerar como un autómata). Cada estado maneja eventos específicos y define las transiciones válidas al siguiente estado. A continuación se enumeran los pasos y estados correspondientes del flujo del proceso.

1. Aceptación de conexión y registro en selector
2. Negociación de método de autenticación → ST_AUTH
3. Autenticación usuario/contraseña, procesamiento de credenciales → ST_AUTH_USERPASS
4. Procesamiento de solicitud de conexión → ST_REQUEST
5. Resolución DNS asíncrona para dominios → ST_RESOLVING
6. Establecimiento de conexión a destino → ST_CONNECTING
7. Transferencia bidireccional de datos → ST_STREAM
8. Limpieza y cierre → ST_DONE

10.3 SISTEMA DE PARSING

Para manejar datos fragmentados, se implementó un sistema de parsing robusto:

- Parser general provisto por la cátedra (*parser.c*)
- Parser específico para autenticación usuario/contraseña, basado en *parser.c* (*userpass_parser.c*)
- Buffers circulares provistos por la cátedra para manejo eficiente de datos (*buffer.c*)
- Validación estricta de límites y formatos

El parser híbrido para autenticación combina la flexibilidad de la máquina de estados general con lógica específica para campos de longitud variable.

10.4 RESOLUCIÓN DNS ASÍNCRONA

Para evitar bloqueos durante resoluciones DNS, se implementó un sistema asíncrono que se lleva a cabo en los siguientes pasos:

1. Creación de thread separado para cada consulta DNS
2. Ejecución de `getaddrinfo()` en el thread
3. Notificación al hilo principal mediante señales
4. Procesamiento del resultado en el contexto del selector

Este diseño mantiene la responsividad del servidor incluso con consultas DNS lentas o que fallan.

10.5 SISTEMA DE MÉTRICAS

Las métricas se organizan en 4 estructuras categorizadas:

- connection_metrics: Estadísticas de conexiones
- transfer_metrics: Datos de transferencia
- performance_metrics: Métricas de rendimiento
- user_metrics: Información de usuarios

La actualización es thread-safe usando mutex, y las consultas proporcionan snapshots consistentes del estado del sistema. Por un lado, el administrador puede visualizar las métricas a tiempo real desde la aplicación cliente, ejecutando el comando `metrics`. Por otro lado, al finalizar la ejecución del servidor, se imprime información sobre la "sesión" del servidor finalizada. Así se ve:

```
=== RESUMEN DE MÉTRICAS SOCKS5 ===
Tiempo de funcionamiento: 18 segundos (0.01 horas)

--- Conexiones ---
Total histórico: 2003
Actuales: 0
Exitosas: 2000
Fallidas: 3
Máximo concurrentes: 13
Por minuto: 60

--- Autenticación ---
Usuarios únicos: 1

--- Transferencia de Datos ---
Total transferido: 5024000 bytes (4.79 MB)
Del cliente: 156000 bytes
Al cliente: 2356000 bytes
Del remoto: 2356000 bytes
Al remoto: 156000 bytes
Throughput actual: 273808.00 bytes/seg (267.39 KB/s)

--- Rendimiento ---
Tiempo promedio de conexión: 17.69 ms
Tasa de éxito: 99.85%
```

10.6 PROTOCOLO DE ADMINISTRACIÓN

El protocolo binario utiliza una estructura simple:

Mensaje de comando:

```
[VERSION:1][COMMAND:1][DATA_LEN:2][DATA:variable]
```

Mensaje de respuesta:

```
[VERSION:1][RESPONSE:1][DATA_LEN:2][DATA:variable]
```

Esta estructura permite extensibilidad futura manteniendo compatibilidad hacia atrás mediante el campo de versión.

10.7 MANEJO DE ERRORES Y ROBUSTEZ

El sistema implementa múltiples capas de manejo de errores:

- Validación de entrada en todos los puntos de entrada
- Manejo graceful de conexiones abortadas
- Recuperación automática de fallos de red
- Logging detallado para debugging
- Cleanup automático de recursos

Para conexiones a dominios con múltiples IPs, se intenta cada dirección secuencialmente hasta encontrar una funcional, proporcionando robustez ante fallos parciales de conectividad.

10.8 CONSIDERACIONES DE SEGURIDAD

Aunque el enfoque principal fue la funcionalidad, se implementaron medidas básicas de seguridad:

- Validación estricta de todos los inputs
- Prevención de buffer overflows
- Separación de privilegios donde es posible
- Logging para futuro análisis de todas las operaciones llevadas a cabo en el servidor
- Manejo seguro de credenciales en memoria

10.9 ESTRUCTURA DE ARCHIVOS

```
src/
├── main.c                      # Punto de entrada principal
├── server/
```

```
│   ├── connection.c/.h         # Manejo de conexiones SOCKS5
│   ├── socks5.c/.h             # Implementación protocolo SOCKS5
│   ├── config.c/.h             # Configuración del servidor
│   └── userpass_parser.c/.h    # Parser autenticación
├── admin_server/
│   ├── admin_protocol.c/.h     # Servidor administración
│   └── admin_config.c/.h       # Configuración admin
├── admin_client/
│   └── admin_client.c          # Cliente administración
├── shared/
│   ├── users.c/.h              # Sistema de usuarios
│   ├── metrics.c/.h            # Sistema de métricas
│   ├── logger.c/.h             # Sistema de logging
│   ├── access_logger.c/.h      # Logging de accesos
│   └── util.c/.h               # Utilidades de red
├── buffer.c/.h                 # Manejo de buffers
├── selector.c/.h               # Multiplexor I/O
├── stm.c/.h                    # Máquina de estados
├── parser.c/.h                 # Sistema de parsing
├── parser_utils.c/.h           # Utilidades de parsing
└── netutils.c/.h               # Utilidades de red
```

Esta organización facilita el mantenimiento y permite modificaciones independientes de cada subsistema.