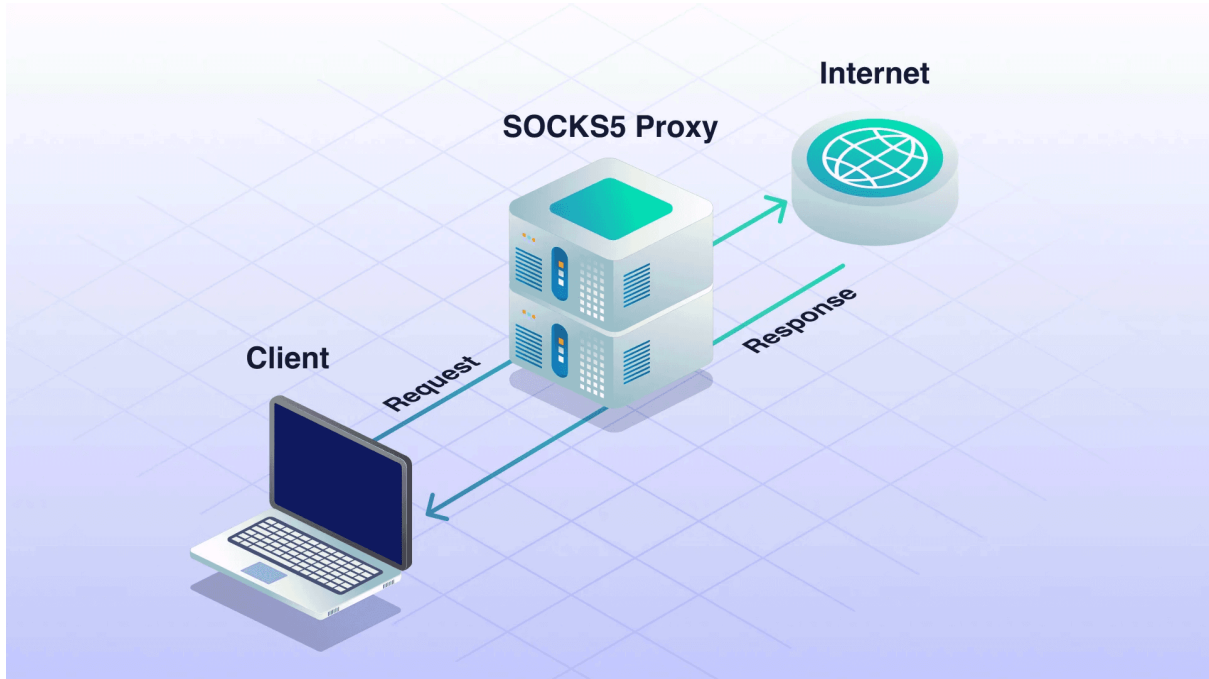


Trabajo Práctico Especial

Protocolos de Comunicación - Grupo 20



legajo	email	integrante
64332	jaliu@itba.edu.ar	Javier Emmanuel Liu
64137	jgago@itba.edu.ar	Juan Diego Gago
64320	akohler@itba.edu.ar	Alex Kenzo Köhler
64500	jbirsa@itba.edu.ar	Juan Pablo Birsa

Profesores

- Marcelo Fabio Garberoglio
- Sebastian Kulesz
- Thomas Mizrahi
- Roberto Oscar
- Hugo Javier
- Pedro Martín Valerio Ramos
- Tomás Ignacio Raiti

Introducción

Este documento presenta el Trabajo Práctico Especial en donde se implementa un servidor proxy siguiendo el protocolo SOCKSv5, donde partimos de los códigos ofrecidos por la cátedra como base. Además, se diseñó e implementó un protocolo de monitoreo para acceder al estado del servidor y configurar usuarios.

Índice

Profesores	1
Introducción	1
Índice	2
Descripción detallada de los protocolos y aplicaciones desarrolladas	2
Problemas encontrados durante el desarrollo	3
Limitaciones de la aplicación	4
Posibles extensiones	4
Conclusiones	4
Ejemplos de Prueba	5
Pruebas de Throughput	5
Prueba de Stress con distintos tamaños de buffer	8
Prueba de Conexión a servicio ipv4 y ipv6	11
Prueba de Direcciones que no responden	12
Prueba de Lecturas/Escrituras parciales	12
Prueba de Pipelining	14
Prueba de Browser	15
Prueba de Integridad	16
Prueba de Autenticación	16
Prueba de Fuzzing	17
Prueba de SIGINT o SIGTERM	18
Guía de instalación	18
Instrucciones para la configuración	19
Documento de diseño del proyecto	20
Protocolo de monitoreo	20
Estado de autenticación	20
Estado de sesión	21
Casos de Error Contemplados	24
Decisiones de diseño en la implementación de socks5	26
Decisiones de diseño en la implementación del Servidor de Monitoreo	26

Descripción detallada de los protocolos y aplicaciones desarrolladas

Se realizó una implementación de un proxy SOCKSv5 siguiendo las especificaciones del RFC 1928. El servidor soporta múltiples pedidos en IPv4 y IPv6 de manera no-bloqueante mediante la multiplexación.

El primer paso es la negociación cliente-servidor, donde el cliente envía los métodos de autenticación que soporta. Nuestro proxy soporta el método sin autenticación y el método de autenticación user/password especificado en el RFC 1929. El proxy elige user/password siempre y cuando el cliente lo soporte, de lo contrario se pasa al método sin autenticación, esto implica que la autenticación user/password es opcional, en la sección **Decisiones de diseño en la implementación de socks5** explicamos las razones de esta decisión.

Una vez autenticado el usuario (si aplica), ya puede enviar el request connect y acceder a distintos origin servers.

Se ha diseñado e implementado un protocolo de monitoreo que facilita el acceso a métricas históricas como el número total de conexiones atendidas, el volumen de datos transferidos y otros indicadores de rendimiento, a través de consultas específicas. El protocolo está documentado en la sección **Protocolo de Monitoreo**.

Utilizamos el registro/logger que propuso la cátedra sin ninguna modificación. El logger por defecto está en modo INFO para imprimir exclusivamente la información pedida por la consigna. Esto implica que los logs nivel DEBUG no se mostrarán.

Problemas encontrados durante el desarrollo

Un problema que fue muy difícil de trackear, fue cuando aborta porque se intentaba mandar datos a un destino que ya cerró la conexión, eso se arregló agregando el flag **MSG_NOSIGNAL** al `send()`.

Nos dimos cuenta usando `strace` como indica la Figura 1

```
sendto(4, "\1", 1, 0, NULL, 0)          = -1 EPIPE (Broken pipe)
--- SIGPIPE {si_signo=SIGPIPE, si_code=SI_USER, si_pid=1404, si_uid=1000} ---
+++ killed by SIGPIPE +++
```

Figura 1: strace dando SIGPIPE

La resolución de dns con thread al principio daba mucho segfault por el mal uso de threads y manejo incorrecto del state machine.

Un problema fue decidir si íbamos a usar binario o texto para el protocolo y cómo parsear el binario. El parser inicial dado por la cátedra no lo tuvimos en cuenta al iniciar a implementar el proyecto.

Dentro del protocolo de monitoreo, una vez nos encontramos dentro del estado de sesión, para la comunicación entre cliente y servidor se definió un protocolo binario en el cual los primeros dos bytes de cada respuesta representan la longitud del mensaje.

Como los protocolos de red TCP/IP utilizan big endian como orden de bytes (también llamado network byte order), y la arquitectura del host (x86/x64) es típicamente little endian, fue necesario usar la función estándar de C **htons**(*host to network short*) para codificar correctamente la longitud antes de enviarla por el socket.

Esta función convierte un número de 16 bits desde el orden de bytes del host (little endian) al orden de red (big endian). Por ejemplo, el número **0x1234** se representa en memoria como **0x34 0x12** en un sistema little endian, y **htons(0x1234)** lo convierte a **0x12 0x34**, como requiere el protocolo de red.

En el cliente, al recibir los datos, fue necesario aplicar la función inversa: **ntohs()** (*network to host short*), para transformar la longitud recibida desde el orden de red a un valor interpretable por la arquitectura del host. Sin esta conversión, la longitud se interpretaría incorrectamente y podría dar lugar a errores como lecturas incompletas o desbordes de buffer.

Limitaciones de la aplicación

El uso de `select()` nos limita a una máxima constante de conexiones simultáneas de 512.

Tamaño de buffer es una constante definida de 32768, osea 32KB, no soportamos cambiarlo en tiempo de ejecución.

Soporta como máximo 216 usuarios, y su nombre y contraseña tienen longitud máxima 64, se rechazará al usuario si tiene más caracteres.

Posibles extensiones

Luego de terminar el trabajo práctico nos dimos cuenta de varias posibles extensiones para el mismo:

- Cambiar el tamaño del buffer en runtime.
- Utilizar un `memory pool` en vez de hacer varios `malloc()` separados. Esto eliminaría el overhead de reservar memoria dinámicamente y simplificará la liberación de memoria.
- Funcionalidades que la consigna no pide, pero el RFC 1928 sí:
 - autenticación GSSAPI
 - método BIND
 - método UDP ASSOCIATE
- Guardar los usuarios en un CSV para que persistan los datos, actualmente no es persistente los usuarios/contraseña.

Conclusiones

En el presente proyecto, tuvimos que encontrar la mejor forma de llevar cada concepto teórico o visto en ejemplos de clase a la práctica durante el cuatrimestre. Logramos que el proyecto funcione en situaciones no-triviales como hacer que el tráfico del navegador Google pase por nuestro proxy de socks 5 sin problemas.

En el proceso de diseñar e implementar el protocolo de monitoreo, logramos crear una definición clara y precisa del protocolo, lo cual llevó a una implementación bastante directa y fácil de probar.

Sería interesante plantear un ejercicio similar al alcanzar la mitad de la cursada, incorporando entonces otros temas estudiados, por ejemplo, UDP mezclado con algún otro aspecto de programación interesante como en este proyecto utilizamos los sockets. Para enriquecer aún más nuestra comprensión y seguir poniendo a prueba nuestras habilidades prácticas.

Ejemplos de Prueba

Las siguientes pruebas se realizaron ejecutando el servidor en localhost puerto 1024 agregando como usuario a `john_doe` con contraseña 1234.

```
./bin/socks5d -p 1024 -P 1025 -u john_doe:1234
```

Pruebas de Throughput

Utilizamos el repositorio <https://github.com/MellanoX/sockperf> para comparar pruebas de sockperf y así obtener distintos Throughput

- Sin el proxy

```

) sockperf pp \
--stream \
-i 192.168.0.243 \
-p 11111 \
-m 1024 \
-t 10 \
--full-log latency_direct.csv \
--histogram 1:0:200

sockperf:
sockperf[CLIENT] send on:sockperf: using recvfrom() to block on socket(s)

[ 0] IP = 192.168.0.243  PORT = 11111 # TCP
sockperf: Warmup stage (sending a few dummy messages)...
sockperf: Starting test...
sockperf: Test end (interrupted by timer)
sockperf: Test ended
sockperf: [Total Run] RunTime=10.000 sec; Warm up time=400 msec; SentMessages=261600; ReceivedMessages=261599
sockperf: ===== Printing statistics for Server No: 0
sockperf: [Valid Duration] RunTime=9.539 sec; SentMessages=249528; ReceivedMessages=249528
sockperf:
sockperf: # dropped messages = 0; # duplicated messages = 0; # out-of-order messages = 0
sockperf: Summary: Latency is 19.066 usec
sockperf: ; each percentile contains 2495.28 observations
sockperf: ---> <MAX> observation = 825.431
sockperf: ---> percentile 99.999 = 341.063
sockperf: ---> percentile 99.990 = 134.957
sockperf: ---> percentile 99.900 = 49.352
sockperf: ---> percentile 99.000 = 31.529
sockperf: ---> percentile 90.000 = 22.006
sockperf: ---> percentile 75.000 = 18.143
sockperf: ---> percentile 50.000 = 17.988
sockperf: ---> percentile 25.000 = 17.893
sockperf: ---> <MIN> observation = 15.068
sockperf: [Histogram] Display scaled to fit on screen (Key: '#' = up to 1559 samples)
sockperf:  bins frequency
sockperf: 15-16 #
sockperf: 16-17 #
sockperf: 17-18 #####
sockperf: 18-19 #####
sockperf: 19-20 ####
sockperf: 20-21 #####
sockperf: 21-22 #####
sockperf: 22-23 ####
sockperf: 23-24 ####
sockperf: 24-25 ###
sockperf: 25-26 ##
sockperf: 26-27 ##
sockperf: 27-28 #
sockperf: 28-29 #
sockperf: 29-30 #
sockperf: 30-31 #
sockperf: 31-32 #
sockperf: 32-33 #

```

Figura 2: throughput sin proxy

- **Con el proxy**

Para la parte del proxy usamos desde nuestra máquina local otra aplicación llamada proxychains4 es una wrapper que fuerza a cualquier aplicación TCP/UDP a pasar todo su tráfico de red a través de uno o varios proxies, seteando que debía pasar de forma estricta por alguno de estas dos dependiendo si lo probamos en una VM o si era en local, nos pareció bastante útil.

```

#socks5 192.168.0.163 1024 john_doe 1234
#socks5 192.168.0.243 1024

```

```

> proxychains4 sockperf pp \
--stream \
-i 192.168.0.243 \
-p 11111 \
-m 1024 \
-t 10 \
--full-log latency_proxy.csv \
--histogram 1:0:200 \
--mps 96000

[proxychains] config file found: /etc/proxychains4.conf
[proxychains] preloading /usr/lib/x86_64-linux-gnu/libproxychains.so.4
[proxychains] DLL init: proxychains-ng 4.16
sockperf:
[proxychains] Strict chain ... 192.168.0.243:1024 ... 192.168.0.243:11111 ... OK
sockperf[CLIENT] send on:sockperf: using recvfrom() to block on socket(s)

[ 0] IP = 192.168.0.243  PORT = 11111 # TCP
sockperf: Warmup stage (sending a few dummy messages)...
sockperf: Starting test...
sockperf: Test end (interrupted by timer)
sockperf: Test ended
sockperf: [Total Run] RunTime=10.000 sec; Warm up time=400 msec; SentMessages=108966; ReceivedMessages=108965
sockperf: ===== Printing statistics for Server No: 0
sockperf: [Valid Duration] RunTime=9.550 sec; SentMessages=104998; ReceivedMessages=104998
sockperf:
sockperf: # dropped messages = 0; # duplicated messages = 0; # out-of-order messages = 0
sockperf: Summary: Latency is 45.371 usec
sockperf: ; each percentile contains 1049.98 observations
sockperf: ---> <MAX> observation = 3289.349
sockperf: ---> percentile 99.999 = 2515.916
sockperf: ---> percentile 99.990 = 903.121
sockperf: ---> percentile 99.900 = 201.313
sockperf: ---> percentile 99.000 = 85.430
sockperf: ---> percentile 90.000 = 55.464
sockperf: ---> percentile 75.000 = 46.286
sockperf: ---> percentile 50.000 = 40.701
sockperf: ---> percentile 25.000 = 39.554
sockperf: ---> <MIN> observation = 37.695
sockperf: [Histogram] Display scaled to fit on screen (Key: '#' = up to 389 samples)
sockperf:  bins frequency
sockperf: 37-38 #
sockperf: 38-39 #####
sockperf: 39-40 #####
sockperf: 40-41 #####
sockperf: 41-42 #####
sockperf: 42-43 #####
sockperf: 43-44 #####
sockperf: 44-45 #####
sockperf: 45-46 #####
sockperf: 46-47 #####
sockperf: 47-48 #####
sockperf: 48-49 #####
sockperf: 49-50 #####
sockperf: 50-51 #####
sockperf: 51-52 #####
sockperf: 52-53 #####
sockperf: 53-54 #####

```

Figura 3: throughput con proxy

También se realizó un programa aparte utilizando python con matplotlib (que se encuentra en `src/test/throughput.py`). El programa crea 512 conexiones simultáneas pidiendo el contenido servido por nginx en localhost (un archivo HTML válido de exactamente 4MB) que pasa por nuestro proxy.

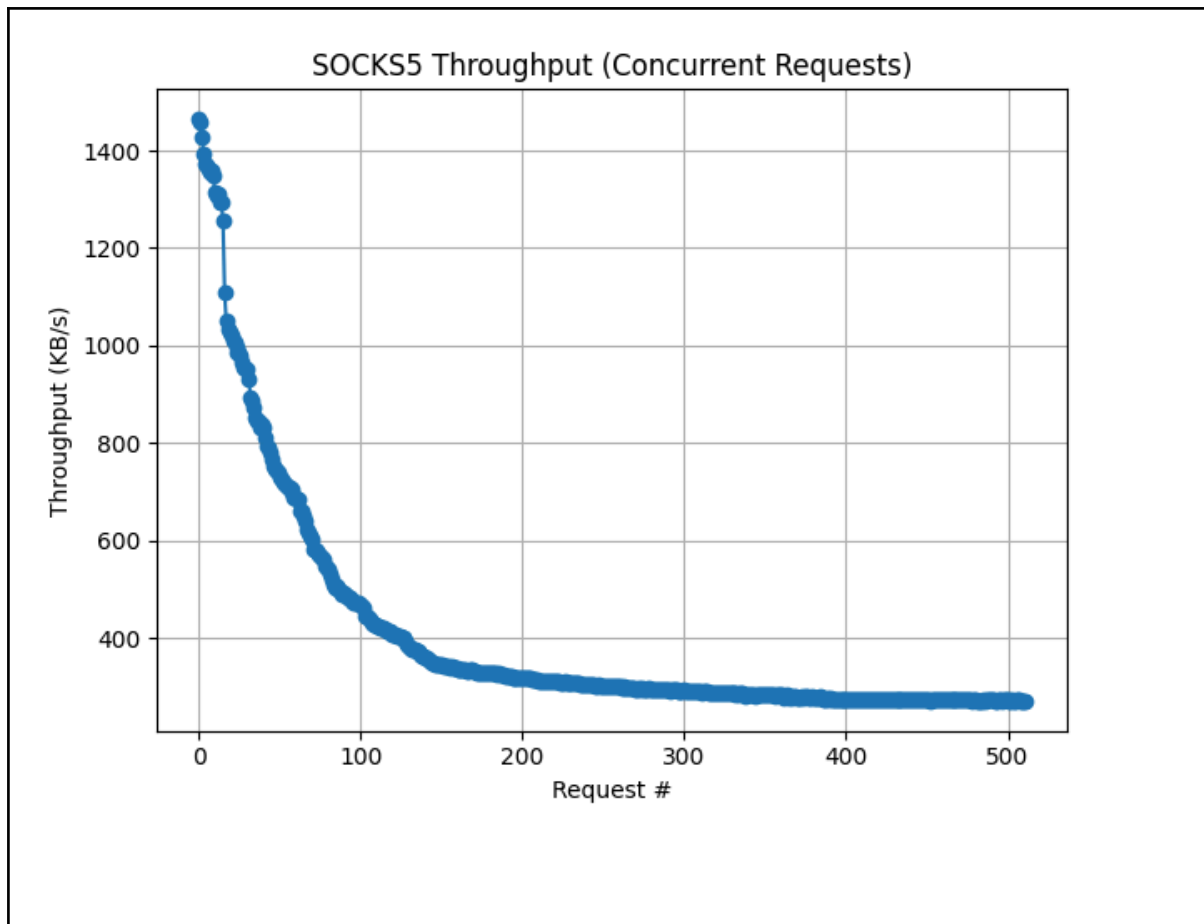


Figura 4.1: cantidad de requests simultáneos vs throughput (KB/s) con proxy

También probamos una modificación leve al código de python para que no pase por el proxy.

En la Figura 4.2 se puede observar que tiene un mejor throughput que en la figura 4.1, notando que las escalas en el eje vertical son distintas. El fenómeno de degradación del throughput se observa en ambos tests, probablemente debido al nginx y limitaciones del hardware.

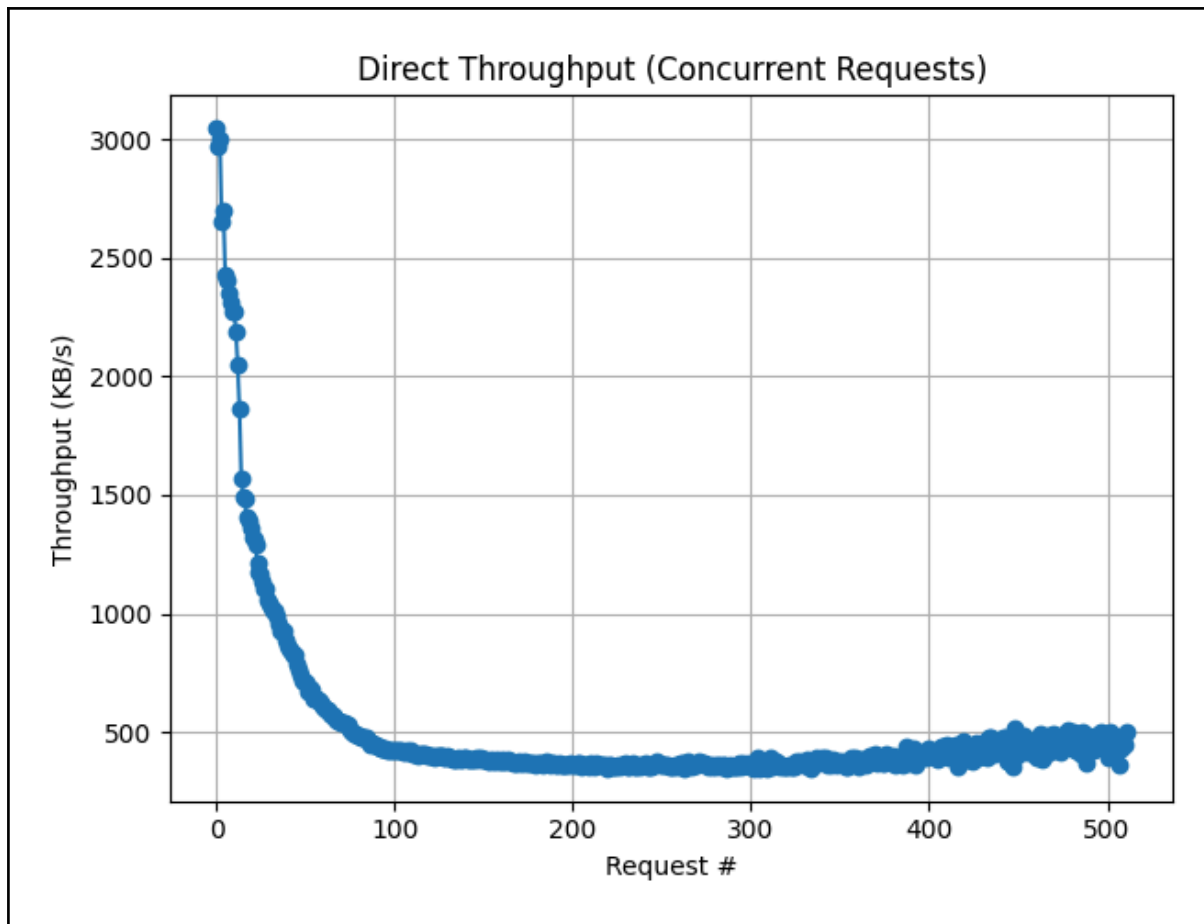


Figura 4.2: cantidad de requests simultáneos vs throughput (KB/s) sin proxy

Conclusión del Throughput

Al emplear nuestro proxy SOCKS5 para el test de throughput, se observa claramente un empeoramiento de los resultados respecto a la conexión directa:

- Mensajes por segundo:
 - Sin proxy: ~249 528 mensajes en 9,539 s → ≈26 152 msg/s.
 - Con proxy: ~104 998 mensajes en 9,550 s → ≈10 999 msg/s.
 Una caída de más del 58 % en la capacidad de envío de mensajes.

- Latencia promedio:
 - Sin proxy: 19,07 μ s (desviación estándar 4,05).
 - Con proxy: 45,37 μ s (desviación estándar 21,76).
 Más del doble de latencia media y una variabilidad sensiblemente mayor (coeficiente de variación pasa de 0,21 a 0,48).

- Distribución de latencias:
 - En la prueba directa, el grueso de las mediciones se concentra entre 17–19 μ s.
 - Con proxy, el histograma se desplaza a 39–41 μ s, y aparecen picos y outliers de hasta varios milisegundos.

En conclusión, el proxy SOCKS5 introduce una sobrecarga significativa: reduce la tasa de mensajes por segundo en más de la mitad, duplica la latencia media y amplía el rango de jitter. Es un resultado esperado dado que el server está corriendo en un solo thread (sin contar los request dns).

Prueba de Stress con distintos tamaños de buffer

Como hemos mencionado anteriormente, una limitación importante de nuestra implementación es que el tamaño del buffer es una constante definida y no soportamos cambiarlo en runtime.

Se creó un server nginx que sirve un archivo en localhost de exactamente 4GB con bytes randomizados generados con el siguiente comando:

```
dd if=/dev/urandom of=random4gb bs=1M count=4096 status=progress
```

Probaremos tamaños de buffer de 4KB, 16KB, 32KB y 64KB pasando por el proxy. Además se probará el tiempo de ejecución sin pasar por el proxy.

```
jaliu@DESKTOP-LHNM420:/var/www/html$ ls -l -h
total 4.1G
-rw-r--r-- 1 root root 4.0G Jul 13 16:29 index.nginx-debian.html
-rw-r--r-- 1 root root 612 Jul 13 16:25 index.nginx-debian.html.old
jaliu@DESKTOP-LHNM420:/var/www/html$ sudo service nginx restart
* Restarting nginx nginx
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl -x socks5h://john_doe:1234@127.0.0.1:1024 localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  892M      0  0:00:04  0:00:04 --:--:--  894M

real    0m4.594s
user    0m0.192s
sys     0m0.932s
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  5622M      0  --:--:--  --:--:--  --:--:--  5618M

real    0m0.735s
user    0m0.088s
sys     0m0.644s
jaliu@DESKTOP-LHNM420:/var/www/html$
```

Figura 5: buffersize de 4KB

```
jaliu@DESKTOP-LHNM420:/var/www/html$ ls -l -h
total 4.1G
-rw-r--r-- 1 root root 4.0G Jul 13 16:29 index.nginx-debian.html
-rw-r--r-- 1 root root 612 Jul 13 16:25 index.nginx-debian.html.old
jaliu@DESKTOP-LHNM420:/var/www/html$ sudo service nginx restart
* Restarting nginx nginx
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl -x socks5h://john_doe:1234@127.0.0.1:1024 localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  1378M      0  0:00:02  0:00:02 --:--:--  1378M

real    0m2.978s
user    0m0.181s
sys     0m0.885s
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  5641M      0  --:--:--  --:--:--  --:--:--  5641M

real    0m0.731s
user    0m0.107s
sys     0m0.621s
jaliu@DESKTOP-LHNM420:/var/www/html$
```

Figura 6: buffersize de 16KB

```
jaliu@DESKTOP-LHNM420:/var/www/html$ ls -l -h
total 4.1G
-rw-r--r-- 1 root root 4.0G Jul 13 16:29 index.nginx-debian.html
-rw-r--r-- 1 root root 612 Jul 13 16:25 index.nginx-debian.html.old
jaliu@DESKTOP-LHNM420:/var/www/html$ sudo service nginx restart
* Restarting nginx nginx
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl -x socks5h://john_doe:1234@127.0.0.1:1024 localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  1673M      0  0:00:02  0:00:02 --:--:-- 1673M

real    0m2.453s
user    0m0.251s
sys      0m1.055s
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  5616M      0  --:--:--  --:--:--  --:--:-- 5610M

real    0m0.734s
user    0m0.064s
sys      0m0.669s
jaliu@DESKTOP-LHNM420:/var/www/html$
```

Figura 7: buffersize de 32KB

```
jaliu@DESKTOP-LHNM420:/var/www/html$ ls -l -h
total 4.1G
-rw-r--r-- 1 root root 4.0G Jul 13 16:29 index.nginx-debian.html
-rw-r--r-- 1 root root 612 Jul 13 16:25 index.nginx-debian.html.old
jaliu@DESKTOP-LHNM420:/var/www/html$ sudo service nginx restart
* Restarting nginx nginx
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl -x socks5h://john_doe:1234@127.0.0.1:1024 localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  1589M      0  0:00:02  0:00:02 --:--:-- 1590M

real    0m2.582s
user    0m0.189s
sys      0m0.834s
jaliu@DESKTOP-LHNM420:/var/www/html$ time curl localhost > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 4096M  100 4096M    0     0  5585M      0  --:--:--  --:--:--  --:--:-- 5580M

real    0m0.739s
user    0m0.111s
sys      0m0.625s
```

Figura 8: buffersize de 64K

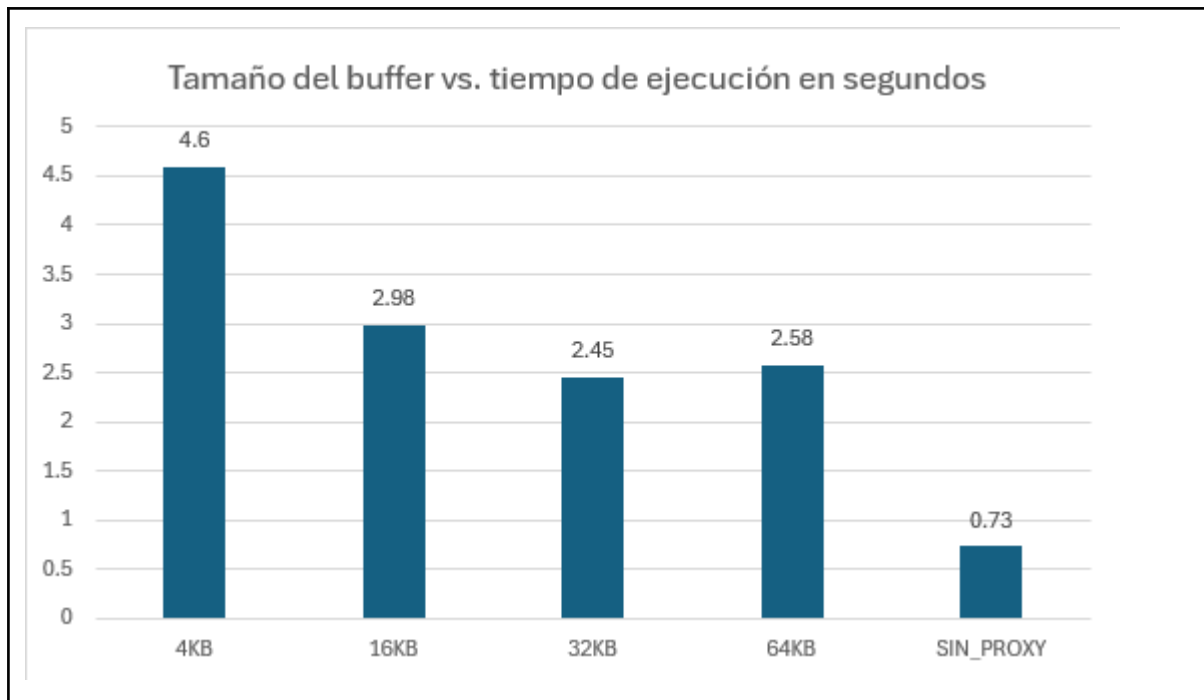


Figura 9: Tamaño de buffer (eje horizontal) vs tiempo en segundos (eje vertical)

La figura 9 es un resumen de los 4 test previos. Donde el eje horizontal indica el tamaño del buffer, excepto el caso donde corremos curl sin el proxy, y el eje vertical que indica el tiempo de ejecución medido en segundos para obtener el archivo de 4GB.

Se puede observar que el tamaño realmente impacta al performance en tamaño 4KB, pero que agregar más tamaño no va a mágicamente mejorar el performance, sino que se observa un efecto de diminishing returns.

A partir de este gráfico, hemos decidido elegir un tamaño de buffer de 32KB, cuyo tiempo de ejecución es x3.35 veces el tiempo que tarda curl sin pasar por proxy, lo cual tiene sentido.

Prueba de Conexión a servicio ipv4 y ipv6

```
jaliu@DESKTOP-LHNM420: ~  
jaliu@DESKTOP-LHNM420:~$ dig example.org +short  
23.215.0.132  
96.7.128.186  
96.7.128.192  
23.215.0.133  
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 -H "Host: example.org" http://23.215.0.132  
<!doctype html>  
<html>  
<head>  
  <title>Example Domain</title>  
  
  <meta charset="utf-8" />  
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />  
  <meta name="viewport" content="width=device-width, initial-scale=1" />  
  <style type="text/css">
```

Figura 10: curl a la ipv4 de example.org

En la figura 10 se puede ver que primero usamos dig para saber la ipv4 de example.org, y después usamos curl a esa ip y retorna lo debido.

Ahora para ipv6, usamos la prueba sugerida por Thomas Mizrahi en los foros.

```
jaliu@DESKTOP-LHNM420:~$ ncat -v -k -l :::1 1234
Ncat: Version 7.80 ( https://nmap.org/ncat )
Ncat: Listening on :::1:1234
Ncat: Connection from ::1.
Ncat: Connection from ::1:48306.
GET / HTTP/1.1
Host: [::1]:1234
User-Agent: curl/7.81.0
Accept: */*
```

Figura 11: ncat escuchando en ipv6 localhost

```
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 127.0.0.1:1234 -v
* Trying 127.0.0.1:1024...
* SOCKS5 connect to 127.0.0.1:1234 (remotely resolved)
* Can't complete SOCKS5 connection to 127.0.0.1. (5)
* Closing connection 0
curl: (97) Can't complete SOCKS5 connection to 127.0.0.1. (5)
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 [::1]:1234 -v
* Trying 127.0.0.1:1024...
* SOCKS5 connect to ::1:1234 (remotely resolved)
* SOCKS5 request granted.
* Connected to (nil) (127.0.0.1) port 1024 (#0)
> GET / HTTP/1.1
> Host: [::1]:1234
> User-Agent: curl/7.81.0
> Accept: */*
>
```

Figura 12: curl al servidor ipv6

En la figura 11 dejamos ncat escuchando en ipv6 localhost puerto 1234.

En la figura 12 primero intentamos mediante el proxy conectarnos al ncat por ipv4, y se rechaza porque no es el protocolo correcto. Después se intenta conectar usando ipv6 y responde correctamente.

Prueba de Direcciones que no responden

En cada prueba, mostraremos un curl sin pasar por el proxy, y otro curl pasando por el proxy.

```
jaliu@DESKTOP-LHNM420:~$ sudo netstat -nltp | grep 6942
jaliu@DESKTOP-LHNM420:~$
jaliu@DESKTOP-LHNM420:~$ curl http://localhost:6942
curl: (7) Failed to connect to localhost port 6942 after 0 ms: Connection refused
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 http://localhost:6942
curl: (97) Can't complete SOCKS5 connection to localhost. (5)
jaliu@DESKTOP-LHNM420:~$
```

Figura 13: Servidor nos informa del rechazo de conexión

Primero verificamos que no haya nada escuchando en el puerto 6969 de localhost

Y después haciendo curl se nos informa que no se pudo completar la conexión con status code 5 (Connection Refused) definido en RFC 1928.

```
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 6.9.6.9:42069
curl: (97) Can't complete SOCKS5 connection to 6.9.6.9. (6)
jaliu@DESKTOP-LHNM420:~$ curl 6.9.6.9:42069
curl: (28) Failed to connect to 6.9.6.9 port 42069 after 134237 ms: Connection timed out
jaliu@DESKTOP-LHNM420:~$
```

Figura 14: Timeout

Hacemos curl a la ip 6.9.6.9 puerto 42069. Después de esperar un tiempo, llega el timeout y se nos informa con el código de error 6 (TTL expired) definido en RFC 1928.

```
jaliu@DESKTOP-LHNM420:~$ dig tejaquiolaip.com +short
jaliu@DESKTOP-LHNM420:~$
jaliu@DESKTOP-LHNM420:~$ curl tejaquiolaip.com
curl: (6) Could not resolve host: tejaquiolaip.com
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 tejaquiolaip.com
curl: (97) Can't complete SOCKS5 connection to tejaquiolaip.com. (4)
jaliu@DESKTOP-LHNM420:~$
```

Figura 15: request a dns inválido

Primero verificamos con dig que el dominio tejaquiolaip.com no retorne una ip. Después con curl vemos que nos informa que no se puede completar la conexión con el status error de 4 (Host unreachable) especificado en RFC 1928.

Prueba de Lecturas/Escrituras parciales

Es un requerimiento del TPE manejar lecturas/escrituras parciales. Es decir, desde el proxy no podemos asumir que el cliente nos va a mandar los paquetes con bytes agrupados exactamente como se especifica en los estados del RFC 1928, sino que se puede mandar de a 1 byte. Para probar eso, se propone el siguiente test en la figura 16:

```
{
    # cliente manda version y metodos que soporta
    sleep 0.3; printf '\x05'
    sleep 0.3; printf '\x03\x00'
    sleep 0.3; printf '\x01\x02'
    # cliente manda usuario y password
    sleep 0.3; printf '\x01\x08\x6a'
    sleep 0.3; printf '\x6f'
    sleep 0.3; printf '\x68\x6e\x5f'
    sleep 0.3; printf '\x64\x6f\x65\x04\x31\x32\x33\x34'
    # cliente manda request a example.org
    sleep 0.3; printf '\x05\x01'
    sleep 0.3; printf '\x00\x03\x0b\x65\x78\x61'
    sleep 0.3; printf '\x6d\x70\x6c\x65\x2e\x6f\x72\x67\x00\x50'
    # HTTP GET
    sleep 0.3; printf 'GET / HTTP/1.1\r\nHost: example.org\r\n\r\n'
} | nc localhost 1024 | hexdump -C
```

Figura 16: mandando paquetes parciales

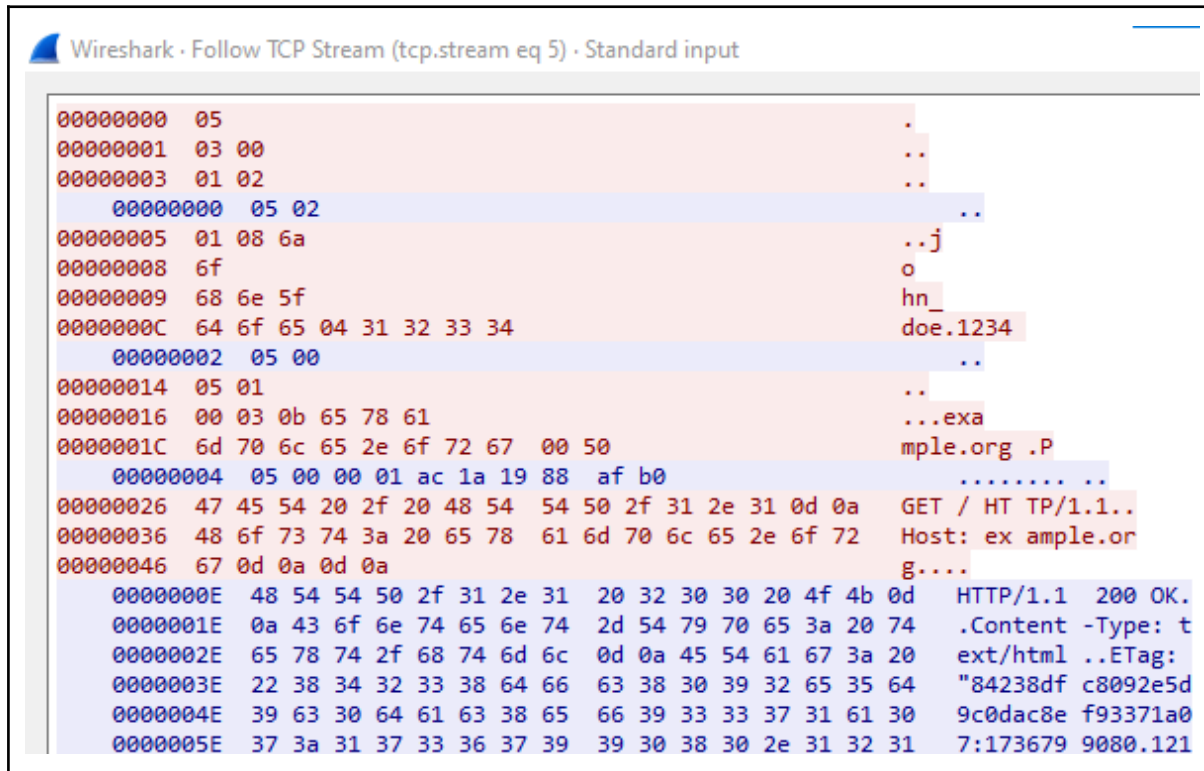


Figura 17: TCP stream de las escrituras parciales

En la Figura 17 los paquetes rojos corresponden al cliente mandando las escrituras parciales (se puede ver separado en distintas líneas, cuya separación coincide con la separación creada en el código de la figura 16).

En azul se ven las respuestas del servidor, manejando la situación como corresponde.

Prueba de Pipelining

Es un requerimiento del TPE manejar pipelining, es decir, que no podemos asumir que el cliente mande los paquetes separados por estado según el RFC 1928, sino que puede mandar todo en un solo paquete y nuestro servidor debe leer hasta lo debido. Propusimos el siguiente test:

```
{
    # cliente manda version y metodos que soporta
    # cliente manda usuario y password
    # cliente manda request a google.com
    sleep 0.3; printf
'\x05\x03\x00\x01\x02\x01\x08\x6a\x6f\x68\x6e\x5f\x64\x6f\x65\x04\x31\x32\x33\x34\x05\x01\x00\x03\x0a\x67\x6f\x6f\x67\x6c\x65\x2e\x63\x6f\x6d\x00\x50'
    # HTTP GET dos veces
    sleep 0.3; printf 'GET / HTTP/1.1\r\nHost: google.com\r\n\r\nGET /
HTTP/1.1\r\nHost: google.com\r\n\r\n'
```

```
} | nc localhost 1024 | hexdump -C
```

Figura 18: mandando los paquetes juntos

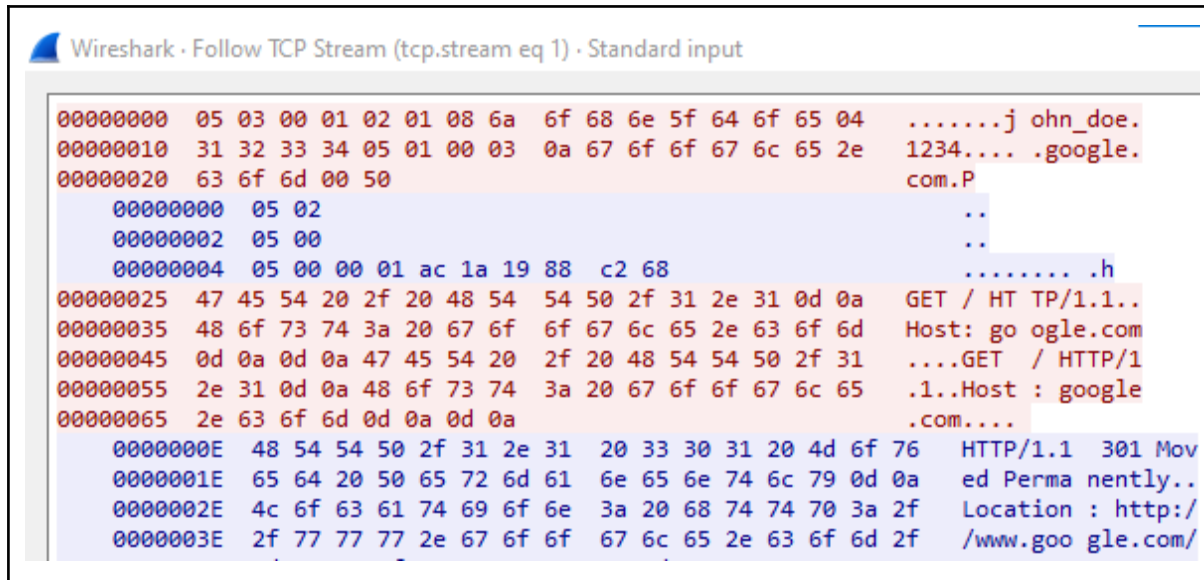


Figura 19: wireshark mostrando el tráfico socks5 pipelining

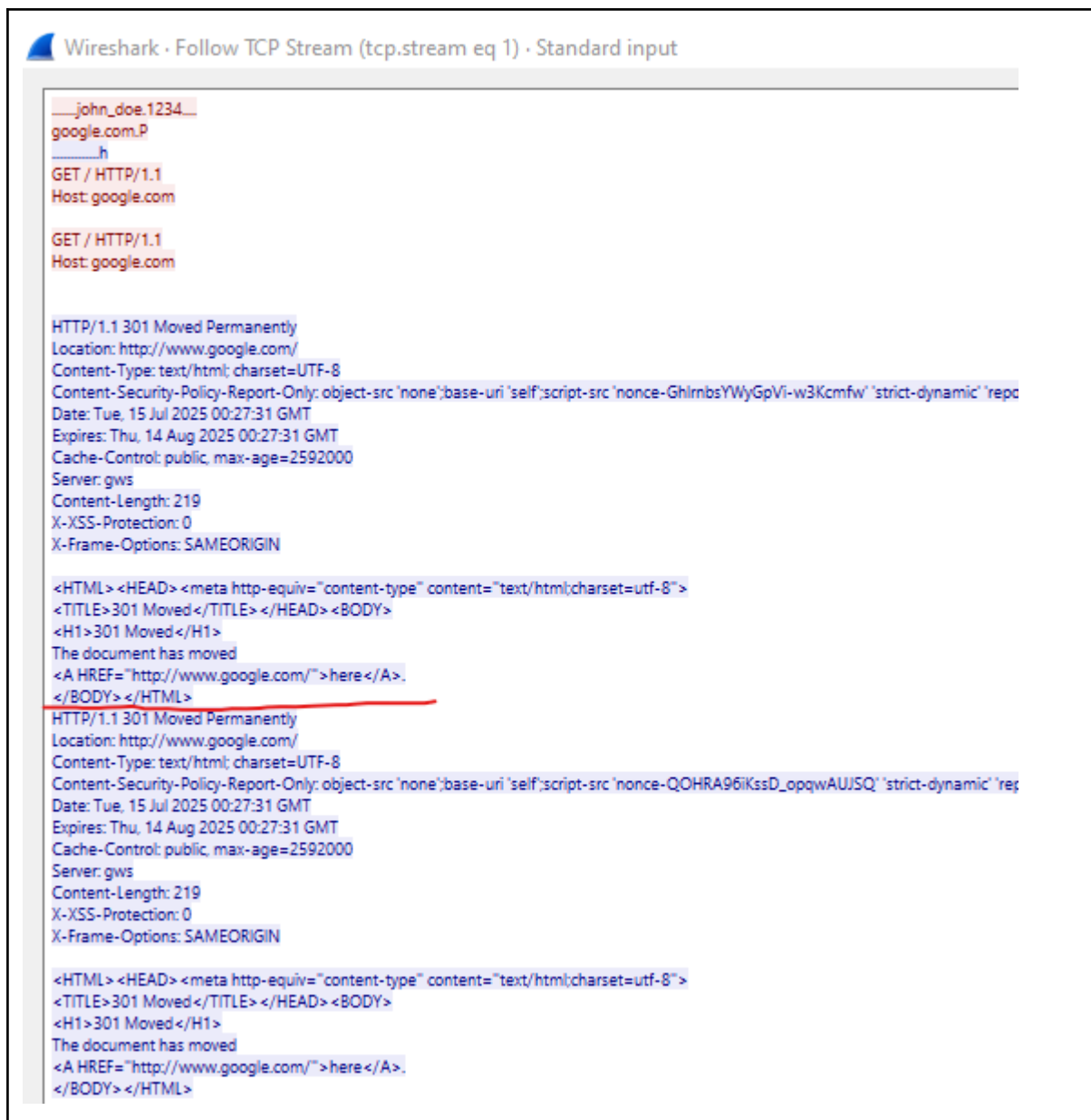


Figura 20: Wireshark modo texto para visualizar HTTP

En el código de la figura 18 se manda en un mismo paquete la versión, las credenciales y el request a [google.com](https://www.google.com). Después se mandan dos GET seguidos.

En la figura 19 se ve el tráfico de socks5 que coincide con lo mandado anteriormente. En la figura 20 se puede observar que el servidor responde correctamente, devolviendo dos veces "MOVED PERMANENTLY", separado por la línea roja.

Prueba de Browser

Corremos google chrome a través de nuestro proxy con el siguiente comando (windows):

```
.\chrome.exe --proxy-server=socks5://127.0.0.1:1024
```

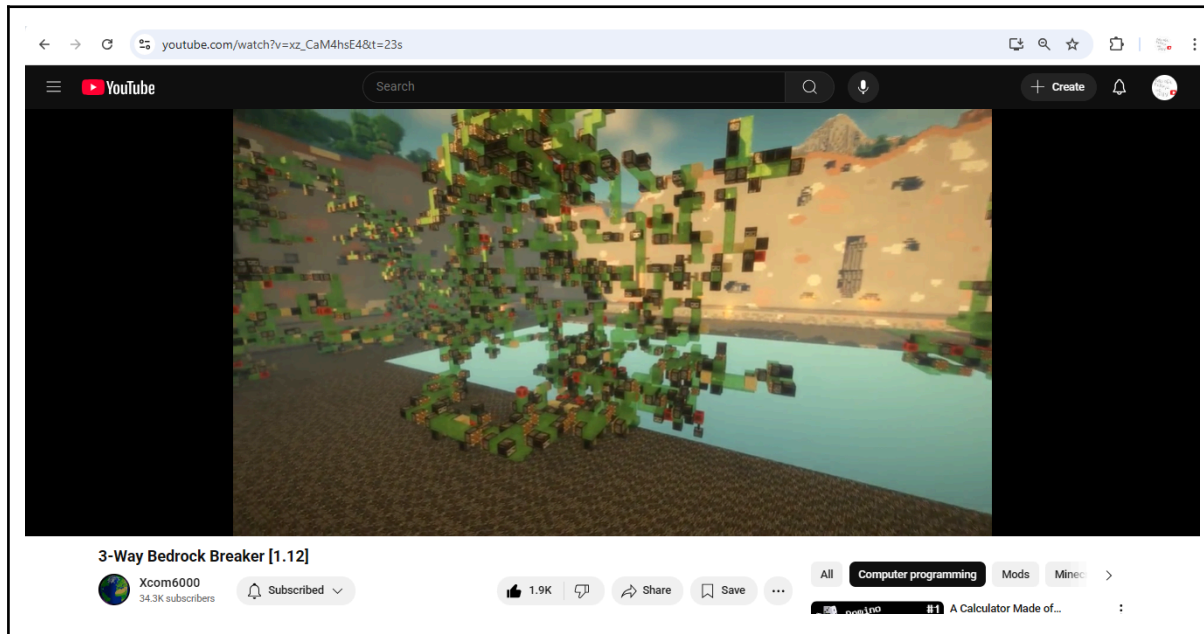


Figura 21: Mirando Youtube a través del proxy

Como se ve en la figura 21, se puede ver Youtube mientras el tráfico está pasando por nuestro proxy. También se probó ir a otras páginas, explorar contenido dinámico, pero eso no lo incluimos en el informe porque necesitaríamos grabar videos.

Prueba de Integridad

```
jaliu@DESKTOP-LHNM420:/var/www/html$ ls -l -h
total 4.1G
-rw-r--r-- 1 root root 4.0G Jul 13 16:29 index.nginx-debian.html
-rw-r--r-- 1 root root 612 Jul 13 16:25 index.nginx-debian.html.old
jaliu@DESKTOP-LHNM420:/var/www/html$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 localhost | md5sum
% Total % Received % Xferd Average Speed Time Time Current
Dload Upload Total Spent Left Speed
100 4096M 100 4096M 0 0 530M 0 0:00:07 0:00:07 --:--:-- 586M
69d351b290eb5cfb97b822aa3660a8b9 -
jaliu@DESKTOP-LHNM420:/var/www/html$ md5sum index.nginx-debian.html
69d351b290eb5cfb97b822aa3660a8b9 index.nginx-debian.html
```

Figura 22: hash md5sum del archivo descargado a través del proxy

Usando el mismo archivo de 4GB introducido en la sección **Prueba de Stress con distintos tamaños de buffer**, comparamos la salida generada por `md5sum` con el original. Los hash generados dan iguales. Esto indica que se transfirieron bien los datos. Anteriormente se probó con `diff`, pero era más lento porque debe guardarse el output en disco.

Prueba de Autenticación

El servidor se inició con un único usuario con nombre "john_doe" y contraseña "1234".

```
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://anon:1234@127.0.0.1:1024 example.org
curl: (97) User was rejected by the SOCKS5 server (5 1).
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:pass@127.0.0.1:1024 example.org
curl: (97) User was rejected by the SOCKS5 server (5 1).
jaliu@DESKTOP-LHNM420:~$ curl -x socks5h://john_doe:1234@127.0.0.1:1024 example.org
<!doctype html>
<html>
<head>
  <title>Example Domain</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <style type="text/css">
body {

```

Figura 23: Prueba de usuarios

En la figura 23 podemos observar tres curl:

1. En el primer curl: se envía un usuario inexistente, y el servidor informa al cliente del curl que se rechazó al usuario.
2. En el segundo curl: se envía un usuario existente, pero con contraseña incorrecta, y el servidor informa al cliente del curl que se rechazó al usuario.
3. En el tercer curl: se envía un usuario y contraseña correctos, y el servidor responde el contenido debido.

Prueba de Fuzzing

Se nos indicó que probarán mandar bytes randomizados al proxy para intentar romperlos (técnica conocida como fuzzing):

```
while true; do
    cat /dev/random | nc localhost 1024
done
```

Figura 24: loop mandando tráfico random al proxy

000151B0	00 cd 5c d0 9f b8 4b df	43 fb d4 7e 5e 50 77 1a	.. \...K. C..~^Pw.
000151C0	80 a8 36 ba 4d 60 40 48	fe eb 9c d5 2e fd 4f 36	..6.M`@H06
000151D0	6f 2d 57 70 d5 3d e4 83	f8 e2 e0 e4 cf a1 8a 12	o-Wp.=..
000151E0	e4 bb ac e0 e4 d5 f0 f5	2a 93 74 a2 7d 6b 95 29 *.t.)k.)
000151F0	57 31 76 1f 4b 52 40 37	60 a7 24 ca 02 e4 0d 47	W1v.KR@7 `.\$....G
00015200	5f 65 dd 72 ca 61 c0 bc	9c 32 f6 ed b9 a9 cb c5	_e.r.a.. .2.....
00015210	d1 0e 69 e6 02 85 56 06	fd 82 61 5e 2c 9e 52 37	..i...V. ..a^,.R7
00015220	dc 4d a5 a6 40 e4 8f 46	56 ad 75 98 4f 3f 2e 59	.M..@...F V.u.O?.Y
00015230	68 0a ae 5e 7e 82 2c e5	b4 0e f4 7d ff 21 67 04	h..^~.,. ...}.!g.
00000000	05 ff		..
00000002	05 01 00 01 00 00 00 00	00 00

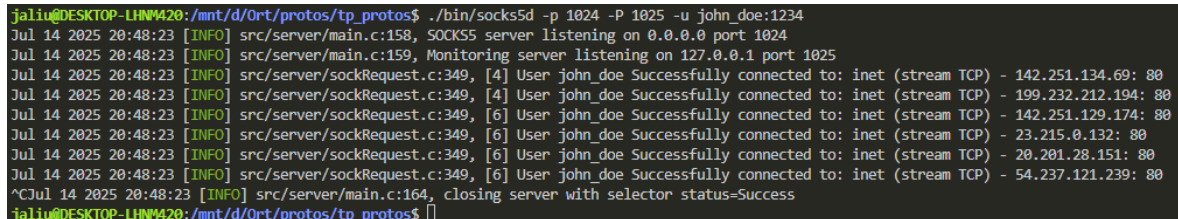
Figura 25: Proxy (azul) rechaza al cliente (rojo) en un Stream específico

Esta prueba es cuestión de correrlo por mucho tiempo, normalmente los random bytes no cumplen el protocolo socks5, y por lo tanto el proxy rechaza al cliente y cierra la conexión.

La parte importante es que no haya segmentation faults o que cause anomalías en el comportamiento del proxy.

Prueba de SIGINT o SIGTERM

Una manera de apagar el proxy socks5 es mandar SIGINT con Ctrl+C



```
jaliu@DESKTOP-LHNM420:/mnt/d/Ort/protos/tp_protos$ ./bin/socks5d -p 1024 -P 1025 -u john_doe:1234
Jul 14 2025 20:48:23 [INFO] src/server/main.c:158, SOCKS5 server listening on 0.0.0.0 port 1024
Jul 14 2025 20:48:23 [INFO] src/server/main.c:159, Monitoring server listening on 127.0.0.1 port 1025
Jul 14 2025 20:48:23 [INFO] src/server/sockRequest.c:349, [4] User john_doe Successfully connected to: inet (stream TCP) - 142.251.134.69: 80
Jul 14 2025 20:48:23 [INFO] src/server/sockRequest.c:349, [4] User john_doe Successfully connected to: inet (stream TCP) - 199.232.212.194: 80
Jul 14 2025 20:48:23 [INFO] src/server/sockRequest.c:349, [6] User john_doe Successfully connected to: inet (stream TCP) - 142.251.129.174: 80
Jul 14 2025 20:48:23 [INFO] src/server/sockRequest.c:349, [6] User john_doe Successfully connected to: inet (stream TCP) - 23.215.0.132: 80
Jul 14 2025 20:48:23 [INFO] src/server/sockRequest.c:349, [6] User john_doe Successfully connected to: inet (stream TCP) - 20.201.28.151: 80
Jul 14 2025 20:48:23 [INFO] src/server/sockRequest.c:349, [6] User john_doe Successfully connected to: inet (stream TCP) - 54.237.121.239: 80
^CJul 14 2025 20:48:23 [INFO] src/server/main.c:164, closing server with selector status=Success
jaliu@DESKTOP-LHNM420:/mnt/d/Ort/protos/tp_protos$
```

Figura 26: SIGINT al proxy

Idealmente, el servidor proxy no debe reportar memory leaks al cerrarlo, como se ve en la figura 26.

Guía de instalación

Para la compilación y ejecución del programa se requiere tener tanto Make como Gcc

Primero para compilar los binarios

› make all

Luego para ver los detalles de los flags y argumentos

› ./bin/socks5d -h

Usage: ./bin/socks5d [OPTION]...

-h	Imprime la ayuda y termina.
-l <SOCKS addr>	Dirección donde servirá el proxy SOCKS.
-L <conf addr>	Dirección donde servirá el servicio de management.
-p <SOCKS port>	Puerto entrante conexiones SOCKS.
-P <conf port>	Puerto entrante conexiones configuracion
-u <name>:<pass>	Usuario y contraseña de usuario que puede usar el proxy. Hasta 10.
-v	Imprime información sobre la versión versión y termina.

por ejemplo

› ./bin/socks5d -p 1024 -P 1025 -u john_doe:1234 -u juan:1234

Para el cliente de monitoreo debes hacer

```
> ./bin/client -h
```

por ejemplo

```
./bin/socks5d -p 2021 -P 2022 -u john:doe
```

Para testear cliente:

```
> curl -x socks5h://john:doe@127.0.0.1:2021 127.0.0.1:80
```

```
> curl -x socks5h://john:doe@127.0.0.1:2021 http://example.org
```

Para wireshark [ALEX]

```
sudo tshark -i lo -w wireshark_captures.pcapng
```

[abrir archivo en wireshark]

Para testear proxy Google Chrome

[Windows 11 con Chrome instalado]

```
> "/mnt/c/Program Files/Google/Chrome/Application/chrome.exe"
```

```
--proxy-server="socks5://127.0.0.1:2021"
```

[En Linux con google-chrome instalado]

```
> google-chrome --proxy-server="socks5://127.0.0.1:2021"
```

Para comparar performance

- Abrir servidor de Python (terminal aparte)
 - python3 ./python_tests/5gb_server.py
- **[CON proxy]**

```
time ALL_PROXY=socks5h://127.0.0.1:2021 curl
http://localhost:8000/ > /dev/null
```
- **[SIN proxy]**

```
time curl http://localhost:8000/ > /dev/null
```

Instrucciones para la configuración

- Cliente de monitoreo

El cliente extrae las credenciales de una variable entorno llamada *MONITORING_TOKEN*.

La misma debe seguir el formato de <user>:<password>. Es importante, por ejemplo antes de correr el programa escribir lo siguiente para cumplir con dicha necesidad de una variable de entorno.

```
> export MONITORING_TOKEN="john_doe:1234"
```

Por último es importante mencionar que hay un administrador el cual será el primero en la lista de usuarios el ejecutar, por ejemplo en el siguiente caso sería "juan".

➤ `./bin/socks5d -p 1024 -P 1025 -u juan:1234 -u nico:1234 -u pedro:coco`

Documento de diseño del proyecto

Protocolo de monitoreo

Dentro de esta sección se detalla el protocolo de monitoreo, cuyo objetivo es mostrar y modificar el estado del servidor.

El protocolo es de tipo binario, ya que al haber implementado el servidor de Socks5 de esta forma, nos encontrábamos familiarizados con este tipo de protocolo.

Cuando el cliente desea hacer uso de este servicio, establece una conexión TCP con el servidor que previamente inició dicho sistema de monitoreo en un puerto puntual, a partir del argumento `-P <número de puerto>`.

Este protocolo tiene 2 estados, en el primer estado, la negociación entre cliente y servidor se basa en el envío de la versión del protocolo, junto a un usuario y contraseña, los cuales fueron previamente creados mediante el argumento `-u <user> <password>`. Como se aclaró anteriormente dentro del informe, el primer usuario dentro de la lista de usuarios creados cuenta con privilegios de administrador, mientras que los demás usuarios solamente cuentan con el rol de **user**. Los privilegios de administrador le permiten al usuario ejecutar ciertos comandos como: cambiar la contraseña de un usuario, cambiar los roles de un usuario, eliminar a un usuario.

En el segundo estado, el servidor y cliente intercambian un comando en particular, dentro de los comandos que ofrece el servidor de monitoreo. Una vez este comando fue intercambiado, la conexión entre cliente y servidor se cierra, por lo cual no es persistente.

Por último, cabe aclarar que el Protocolo de Monitoreo se encuentra diseñado para leer paquetes fragmentados, esto mismo puede ser *testado* a partir de unos tests, escritos en el lenguaje python, que se encuentran dentro de la carpeta `python_tests`.

Estado de autenticación

Dentro de este estado, el usuario se va a autenticar, para luego poder acceder a los servicios que ofrecemos dentro del servidor. En un principio los usuarios existentes dentro del servidor son creados al momento de ejecutar el archivo `./bin/socks5d`, como se detalla en la sección: [Guía de instalación](#). Y además, las credenciales de `username` y `password` se obtienen a partir de la variable de entorno `MONITORING_TOKEN`, como se detalla también dentro de la sección: [Instrucciones para la configuración](#).

Cliente envía

NOMBRE DE CAMPO	VER	ULEN	USERN	PLEN	PASSWD
CANTIDAD BYTES	1	1	1 to 64	1	1 to 64

Servidor responde

NOMBRE DE CAMPO	VER	STATUS
CANTIDAD BYTES	1	1

El campo de versión se encuentra, en el caso de una posible extensión del protocolo, actualmente esta versión es la 1. Mientras que el campo status va a contener un “01” en el caso de autenticación exitosa, y un “00” en caso contrario.

Estado de sesión

Para la comunicación entre cliente y servidor se definió un protocolo binario en el cual los primeros dos bytes de cada respuesta del servidor representan la longitud del mensaje, seguido por el mensaje respectivo.

NOMBRE DE CAMPO	ANSLEN	ANS
CANTIDAD BYTES	2	1 to 65534

Por otro lado, dentro del envío por parte del cliente, el primer byte indica qué tipo de comando se va a enviar, por ejemplo dentro del comando **LIST USERS**, se envía un 1, mientras que para **ADD USER** se envía un 2, y así sucesivamente. Mientras que en el caso de que el comando requiera más información, como por ejemplo un *username*, o una *password*, esta misma se va a enviar como todo protocolo de tipo binario, el primer byte indica la longitud, seguido del string.

LIST USERS

Este comando se encarga de listar los usuarios actuales del servidor.

```

jbirsa@DESKTOP-FV3HC0A:~/projects/PROTOS/tp_protos$ ./bin/client 1025 LIST USERS
Registered users:
admin
carlitos
carlos
john doe

```

Cliente envía:

NOMBRE DE CAMPO	CMD
CANTIDAD BYTES	1

Dentro de CMD, el byte que se envía contiene el valor correspondiente al comando, 1.

ADD USER <user> <password>

Este comando se encarga de agregar un usuario al servidor. El rol de este mismo, por defecto es USER.

```
jbirsa@DESKTOP-FV3HC0A:~/projects/PROTOS/tp_protos$ ./bin/client 1025 ADD USER john_doe 123
User john_doe added successfully
Registered users:
admin
carlitos
carlos
john_doe
```

Cliente envía:

NOMBRE DE CAMPO	CMD	ULEN	UNAME	PLEN	PASSWD
CANTIDAD BYTES	1	1	1 to 64	1	1 to 64

Dentro de CMD, el byte que se envía contiene el valor correspondiente al comando, 2. Mientras que el *username* y *password* es el correspondiente a agregar al servidor.

REMOVE USER <user>

Este comando se encarga de eliminar a un usuario que se encuentra dentro del servidor. Solamente puede ser ejecutado por un usuario con roles de ADMIN.

```
jbirsa@DESKTOP-FV3HC0A:~/projects/PROTOS/tp_protos$ ./bin/client 1025 REMOVE USER john_doe
User john_doe deleted
Registered users:
admin
carlitos
carlos
```

Cliente envía:

NOMBRE DE CAMPO	CMD	ULEN	UNAME
CANTIDAD BYTES	1	1	1 to 64

Dentro de CMD, el byte que se envía contiene el valor correspondiente al comando, 3. Mientras que el *username* es el correspondiente a eliminar del servidor.

CHANGE PASSWORD <user> <password>

Este comando se encarga de cambiar la contraseña de un usuario que se encuentra dentro de nuestro servidor. Solamente usuarios con privilegios de ADMIN pueden ejecutarlo.

```
jbirsa@DESKTOP-FV3HC0A:~/projects/PROTOS/tp_protos$ ./bin/client 1025 CHANGE PASSWORD carlos 1234567
Password successfully changed for user carlos.
Registered users:
admin
carlitos
carlos
```

Cliente envía:

NOMBRE DE CAMPO	CMD	ULEN	UNAME	PLEN	PASSWD
CANTIDAD BYTES	1	1	1 to 64	1	1 to 64

Dentro de CMD, el byte que se envía contiene el valor correspondiente al comando, 4. Mientras que el *username* y *password* es el correspondiente a cambiar la contraseña dentro del servidor.

GET METRICS

Este comando se encarga de mostrar las métricas actuales de nuestro servidor de Socks5, las métricas que consideramos necesarias para nuestro servidor fueron solamente las indispensables:

- Cantidad de conexiones históricas
- Cantidad de conexiones concurrentes (Conexiones actuales al momento de ejecutar el comando)
- Cantidad de bytes enviados por el servidor Socks5
- Cantidad de bytes recibidos por el servidor Socks5

```
jbirsa@DESKTOP-FV3HC0A:~/projects/PROTOS/tp_protos$ ./bin/client 1025 GET METRICS
Total connections: 20
Current connections: 0
Bytes sent: 32152
Bytes received: 32597
```

Cliente envía:

NOMBRE DE CAMPO	CMD
CANTIDAD BYTES	1

Dentro de CMD, el byte que se envía contiene el valor correspondiente al comando, 5.

CHANGE ROLE <user> <role>

Este comando se encarga de cambiar los roles de los usuarios que se encuentran dentro de nuestro servidor. Solamente puede ser ejecutado por usuarios con privilegios de ADMIN.

```
jbirsa@DESKTOP-FV3HC0A:~/projects/PROTOS/tp_protos$ ./bin/client 1025 CHANGE ROLE carlos 1
Role successfully changed for user carlos to ADMIN.
Registered users:
admin
carlitos
carlos
```

Cliente envía:

NOMBRE DE CAMPO	CMD	ULEN	UNAME	ROLE
CANTIDAD BYTES	1	1	1 to 64	1

Dentro de CMD, el byte que se envía contiene el valor correspondiente al comando, 6. Mientras que el *username* y *role* es el correspondiente a cambiar de rol dentro del servidor.

Casos de Error Contemplados

Dentro de esta sección, se muestran los posibles casos de error dentro del Protocolo de Monitoreo.

- Contraseña incorrecta

```
> export MONITORING_TOKEN="john_doe:1234"
> ./bin/client 1025 LIST USERS
Registered users:
john_doe
juan
> export MONITORING_TOKEN="john_doe:1235"
> ./bin/client 1025 LIST USERS
Client error: incorrect password
user auth failed
```

- Listado de usuarios:

```
> ./bin/client 1025 LIST USERS
Registered users:
john_doe
juan
nico
```

- Agregar un usuario

```
> ./bin/client 1025 LIST USERS
Registered users:
john_doe
juan
nico
> ./bin/client 1025 ADD USER Alex 123456

User Alex added succesfully
Registered users:
Alex
john_doe
juan
nico
> export MONITORING_TOKEN="Alex:123456"
> ./bin/client 1025 LIST USERS
Registered users:
Alex
john_doe
juan
nico
```

- Remover un usuario

```
> export MONITORING_TOKEN="nico:1234"
> ./bin/client 1025 REMOVE USER john_doe
Error: Only admins can remove users
> export MONITORING_TOKEN="john_doe:1234"
> ./bin/client 1025 REMOVE USER john_doe
Error: can't delete john_doe because is the last admin
```

- Token Inválido

```
> ./bin/client 1025 GET METRICS

No token provided for connection
El servidor cerró la conexión
```

- Cambiar la contraseña de un usuario

```
> export MONITORING_TOKEN="nico:1234"
> ./bin/client 1025 CHANGE PASSWORD john_doe 123456
Error: Only admins can change passwords
> export MONITORING_TOKEN="john_doe:123456"
> ./bin/client 1025 CHANGE PASSWORD fake_name 123456
Error: User fake_name does not exist
```

- Cambiar el rol de un usuario

```
> export MONITORING_TOKEN="nico:1234"
> ./bin/client 1025 CHANGE ROLE john_doe 0
Error: Only admins can change roles
> export MONITORING_TOKEN="john_doe:1234"
> ./bin/client 1025 CHANGE ROLE john_doe 0
Error: Cannot demote john_doe as they are the last admin
```

- Si ponemos un ADD USER con un username muy largo

```
> ./bin/client 1025 ADD USER usernameasdasdasdasdbaksdbaskjdbaskjbakjfbbsadlfbsal
dfbashdfbsaldjfbhsadljfhasdbfhkbasdlfbsadfhlabsdf password
Client error: username length is invalid
Error sending ADD USER command
```

Decisiones de diseño en la implementación de socks5

Cuando hay un error, y el RFC 1928 no especifica, entonces se manda error general X"01".

Por otro lado, proveemos dos métodos de autenticación:

1. sin autenticación
2. autenticación con usuario/password

Se aceptan pedidos sin autenticación debido a que configurar la autenticación puede llegar a ser tedioso, especialmente en Google Chrome.

Si el cliente ofrece el método de autenticación con usuario/password, entonces el proxy elegirá usuario/password. De lo contrario elegirá "sin autenticación".

Decisiones de diseño en la implementación del Servidor de Monitoreo

Dentro del servidor de monitoreo, no se utilizó el *parser* dado por la cátedra, dado que nos pareció más fácil utilizar uno propio, ya sea para la parte del cliente, como para la parte del servidor.

Dentro de la parte del servidor, no hizo falta utilizarlo, dado que por el diseño que tomamos para el protocolo, el primer byte nos indica el tipo de comando a utilizar, por lo cual fue bastante sencillo obtener el tipo de comando. Mientras que si necesitamos otro tipo de dato, el primer byte nos indica la longitud de este mismo, por lo cual sabemos hasta cuando tenemos que "leer" del *buffer*.

Por otro lado, por parte del cliente, utilizamos un *parser* propio para obtener el tipo de comando que debíamos ejecutar. Y para esto mismo, solamente tuvimos en cuenta los argumentos y la cantidad de los mismos. Por ejemplo, en el caso de ejecutar el comando *LIST USERS*, solamente esperamos 4 argumentos (los otros dos el nombre del archivo a ejecutar y el número de puerto al que nos debemos conectar), y que los argumentos 3 y 4 sean LIST y USERS respectivamente.

```
jbirsa@DESKTOP-FV3HC0A:~/projects/PROTOS/tp_protos$ ./bin/client 8081 LIST USERS
```

Por último, cabe aclarar que al momento de *parsear* los datos que obtenemos por la entrada, con el objetivo principal de **poder leer paquetes fragmentados**, no utilizamos específicamente las funciones dentro de *parser.c*, pero obtuvimos una inspiración a partir de esto mismo. Básicamente lo que hacemos es leer a partir de una variable dentro del struct *MonitoringClientData*, la cual se llama *toRead*. Esta misma variable nos indica la cantidad de bytes que se encuentran disponibles para leer, mientras que la variable *parsing_state* se encarga de indicar que se debe leer (username length, y luego contraseña, etc.).