



Trabajo Práctico Especial

17/11/2020

Geronimo Máspero

58322

Scott Lin

59339

Nicolás Comerci

59520

Martín Ciccioli

58361

Introducción	2
Protocolos y aplicaciones desarrolladas	2
SOCKSv5	2
LOGGING	4
SNIFFING	4
DNS mediante HTTP (DoH)	5
<i>Consulta DoH</i>	5
<i>Respuesta DoH</i>	6
Protocolo G8	7
Negociación inicial	7
GET	8
SET	9
QUIT	12
Decisiones sobre el protocolo	13
Implementación en el proxy	13
Problemas encontrados durante el diseño y la implementación	14
Limitaciones de la aplicación	14
Posibles extensiones	14
Conclusiones	15
Ejemplos de prueba	16
Guía de instalación detallada y precisa	21
Instrucciones para la configuración	21
Ejemplos de configuración y monitoreo	22
Cliente	22
Diagrama de diseño del proyecto	26

Introducción

El objetivo de este trabajo práctico es implementar un servidor proxy para el protocolo SOCKSv5 basándose en el *RFC 1928*. El servidor debe contar con las siguientes funcionalidades:

- Debe atender a múltiples clientes en forma concurrente.
- Los clientes deben poder autenticarse utilizando un usuario y una contraseña como indica el *RFC 1929*.
- Debe soportar consultas a direcciones IPv4, IPv6 o FQDN.
- Debe contar con un mecanismo que ayude a monitorear la operación del sistema y permite cambiar la configuración del servidor en tiempo de ejecución.
- Debe mantener un registro de acceso que permita dar a conocer quien se conectó, a qué sitio y cuándo.
- Debe poder monitorear el tráfico y generar un registro de credenciales de acceso para los protocolos HTTP y POP3.

Protocolos y aplicaciones desarrolladas

SOCKSv5

Basándonos *RFC 1928*, se define el protocolo SOCKSv5 como un framework que permite atravesar de forma segura y transparente firewalls que aíslan una red. Siguiendo esta definición, implementamos nuestro servidor proxy SOCKSv5 y de manera no bloqueante que permite un máximo de 1024 conexiones concurrentes, tanto IPv4 como IPv6.

La conexión entre el cliente y el proxy se realiza en distintos pasos, primero se envía un paquete de identificación del protocolo y de los métodos de autenticación del usuario, que llamamos *'HELLO'*. Luego, el servidor elige el método de autenticación que desea utilizar y en caso de no ponerse de acuerdo con el usuario, le responde con el mensaje correspondiente y cierra la conexión. En caso contrario, y si se especificó un método de autenticación, el cliente envía sus credenciales para autenticarse con el servidor. Finalmente, si no se especificó un método de autenticación o si las credenciales son correctas, el cliente envía su pedido de conexión (podrían ser otros comandos pero en

nuestra implementación solo permitimos connect) y el servidor intenta crear la conexión con el servidor de origen especificado.

Para lograr dicho objetivo, el proxy puede tomar dos caminos:

- En caso de que el cliente haya especificado una dirección IP, se intentará establecer la conexión. Si la misma es exitosa o no, se le informa al cliente utilizando mensajes adecuados establecidos por el *RFC 1928*.
- En caso de que el cliente haya especificado un FQDN, se realiza una consulta DOH (DNS over HTTP) a un servidor previamente establecido pidiendo las direcciones IPv4 de dicho dominio. Una vez obtenida la respuesta, se intenta establecer conexión con la primera dirección recibida. Si la misma es exitosa, se le informa al cliente y sino se repite el proceso con las demás direcciones. En caso de que no haya sido posible la conexión con ninguna de ellas, se repite la consulta DOH pero esta vez pidiendo las direcciones IPv6. Con la respuesta recibida, se repite el proceso anteriormente mencionado. Si así todo no se pudo conectar, se le informa al cliente que el host no es accesible.

Todos los pedidos de conexión, junto con su resultado, quedan registrados en la salida estándar (ver sección LOGGING).

Una vez lograda la conexión con el servidor de origen, los paquetes enviados por el cliente pasan por el proxy, se redirigen al origen y viceversa.

Para lograr todo lo dicho, siendo que el proxy se ejecuta en un único hilo, se utilizó un selector donde se registran los file descriptors de cada socket abierto y es el encargado de iterar por todos ellos para realizar las operaciones correspondientes sólo cuando son requeridas.

Adicionalmente, el proxy tiene implementado un sniffer de contraseñas para paquetes HTTP y POP3 que son explicados en detalle en la sección SNIFFING

Dicha opción, junto con otras, puede ser modificada en tiempo de ejecución del proxy mediante un protocolo de configuración para administradores. El mismo es explicado en detalle en la sección PROTOCOLO G8

Finalmente, cuando el cliente o el servidor origen cierra la conexión, se le informa al lado correspondiente y se liberan todos los recursos reservados.

Si el administrador decide cerrar el proxy, previamente se cierran todas las conexiones abiertas y se liberan todos los recursos.

LOGGING

Para guardar información acerca de las conexiones establecidas a través del proxy, estas se registran en pantalla (STDOUT), con información acerca de la fecha (en formato ISO-8601) en la que se estableció, el usuario del servidor que generó la conexión (en caso de que no haya usuario, se imprime "----"), su dirección IP y puerto y la dirección IP y puerto al que se conectó.

Ejemplo de conexión sin autenticación al puerto 80 de google.com:

```
[2020-11-14T03:13:28Z] ---- A ::1 43760 www.google.com 80
status=0
```

Al ser un servidor que atiende múltiples conexiones de manera constante, no podemos permitir que se bloquee, por ejemplo escribiendo en el fd de STDOUT, por lo que se implementó la escritura en STDOUT de manera no bloqueante. Esto se hizo registrando el fd asociado al STDOUT en el selector como OP_NOOP y asociándole un buffer dentro del cual se hace la escritura de lo que se quiere enviar a STDOUT. Cuando se escribe en dicho buffer, se cambia el interés del fd a OP_WRITE para informarle al selector que nos interesa escribir a STDOUT y, finalmente, se escribe en STDOUT cuando estamos seguros de que esta acción no nos va a bloquear, ya que nos lo garantiza el select. Una vez hecha la escritura y si quedó algo por escribir, no se cambian los intereses del fd; caso contrario, se vuelve a poner en OP_NOOP.

SNIFFING

Además de guardar información acerca de las conexiones que pasan por el proxy, también se guarda información acerca de las credenciales que pasan por el mismo. Nuestra implementación soporta sniffing de credenciales en HTTP y POP3. Esto se logró parseando los paquetes correspondientes que pasaban por el proxy.

Primero si el paquete en cuestión pertenece a alguno de estos protocolos. Para dicha tarea, se utilizan algunas heurísticas: como por ejemplo, que el primer paquete enviado en HTTP es del cliente y se debe encontrar la palabra "HTTP" antes del primer "\r\n". En el caso de POP3, se sabe que el primer paquete enviado es del servidor de origen y que su primer mensaje contiene la palabra "+OK".

Una vez identificado el paquete, se parsea hasta llegar al mensaje que nos interesa. En el caso de HTTP, se busca en el header, la sección "Authorization" que contiene el tipo de autenticación que se utiliza para codificar las credenciales junto con el nombre de usuario y las contraseña codificadas. En nuestro caso, solo soportamos sniffing para el tipo Basic, que codifica las credenciales con Base64. Estas credenciales se decodifican y se imprimen en STDOUT mediante la función de logging. Si el parser llega al final del header y no encuentra lo anteriormente mencionado, detiene el parseo para el resto del paquete y todos los posteriores.

En el caso de POP3, buscamos los mensajes "USER" y "PASS" que son los que utiliza el usuario para identificarse ante el servidor. Luego se parsea la respuesta del origen para chequear si las credenciales son correctas. Si no lo son se sigue parseando y, en caso contrario, se imprimen en STDOUT y se detiene el parseo.

Al igual que se vio en la sección LOGGING, el resultado del sniffing se imprime en STDOUT de manera no bloqueante de la misma manera que para los registros de acceso. Si bien ambos de estos protocolos implementan el pipelining, nuestra implementación de sniffers no es capaz de obtener las credenciales enviadas con este método.

DNS mediante HTTP (DoH)

DoH es un protocolo de seguridad para realizar una resolución remota del sistema de nombres de dominio (DNS) a través del protocolo HTTP. Este método tiene como finalidad mejorar la seguridad del usuario. De esta manera, cuando el usuario realiza una consulta DNS, en vez de operar con UDP, lo hace con mayor protección a través de TCP.


La consulta HTTP puede ser llevada a cabo de dos maneras distintas. La misma puede realizarse mediante el método GET, donde la consulta DNS va codificada como un parámetro de la URL, o mediante el método POST, donde la consulta DNS va en el "body" del request HTTP. Optamos por implementar la primera opción ya que el método GET es cacheable, evitando interacción recurrente con el servidor DNS en casos innecesarios y optimizando el uso de los recursos.

Consulta DoH

En primer lugar, generamos la consulta DNS. Basándonos en el *RFC1035* y en el uso de la herramienta Wireshark, implementamos una función que, al pasarle como argumentos un cierto FQDN y un type, produce una consulta DNS para dicho dominio. La consulta DNS está compuesta por dos secciones: el header y la query.

En primer lugar, para implementar el header, ejecutamos una consulta DNS preguntando sobre un cierto dominio y analizamos con atención el paquete de la consulta capturado por Wireshark. De esa manera, logramos estudiar los bytes que se envían en el header al realizar la consulta DNS. Tomamos estos bytes como base para formar la estructura de nuestra consulta; los únicos bytes que modificamos fueron el Transaction ID (lo pusimos en 0 ya que no es relevante para nuestro caso, dado que estamos realizando las consultas a través de TCP y no UDP) y el Additional Records (lo seteamos en 0 ya que en nuestro caso no usamos registros adicionales).

En segundo lugar, para implementar la query, nos basamos en los tres campos descriptos por el RFC mencionado anteriormente: QNAME, TYPE y CLASS. Para el QNAME, desarrollamos un algoritmo que transforma el FQDN que le enviamos a un formato



especial que es compatible con lo especificado por el RFC. Con respecto al campo TYPE, estudiamos el argumento recibido y conforme a si el mismo es IPV4 o IPV6, seteamos el último byte de este campo en 0x01 o 0x1C respectivamente. Por último, el campo CLASS se mantiene fijo ya que siempre estamos preguntando por un IN.

Luego, una vez finalizada la estructura de la consulta DNS, compuesta por el header y el query, codificamos la misma en BASE64 URL. Esto es imprescindible ya que es requisito de compatibilidad de la consulta HTTP del protocolo DoH. Luego, para finalizar la consulta DoH, utilizamos el encoding obtenido en el paso anterior y lo combinamos con ciertos “strings” que deben ir por defecto en la consulta, los cuales definimos en base al *RFC8484*.

Respuesta DoH

Para procesar la respuesta DoH, volvimos a basarnos en el *RFC8484*, y desarrollamos dos parsers: uno para extraer la información relevante del header, y otro para procesar el body.

En primer lugar, en el parser del header, verificamos que los headers imprescindibles en la respuesta DoH estén presentes. Corroboramos que el status code sea el esperado, y que tanto el header Content-type como Content-Length estén presentes y tengan el contenido adecuado. En caso que se cumplan estos requisitos y el parser lea un “\r\n\r\n”, se pasa al parser del Body. En cambio, si la validación falla, el parser retorna con un error.

En segundo lugar, desarrollamos un parser para extraer las respuestas del body. El body contiene la respuesta DNS, la cual cuenta con tres secciones: header, query y answer. Tanto el header como el query coinciden con la consulta DNS que mencionamos en el apartado anterior (“*Consulta DoH*”). Aprovechando esto, utilizamos la longitud de la consulta DNS generada anteriormente para navegar sobre los bytes del body y así llegar a la sección answer, que es la que nos interesa. Sin embargo, como la cantidad de respuestas en la sección answer puede variar, antes de llegar a la misma leemos los dos bytes del header DNS que representan la cantidad de respuestas que hay disponibles. Utilizando este valor y un contador, leemos las respuestas disponibles y las almacenamos en un arreglo.

Protocolo G8

Este protocolo busca permitir la obtención de métricas de uso de un proxy SOCKSv5 y la modificación de la configuración del servidor en runtime. Es un protocolo orientado a sesión y funciona sobre SCTP. Permite 3 tipos de comandos, GET, SET, QUIT. El primer comando permite obtener algún dato almacenado en el servidor. El siguiente permite modificar alguna configuración del servidor o información acerca de los usuarios/admins. Y finalmente, el último permite el cierre de la conexión.

Negociación inicial

El usuario genera el primer pedido de conexión enviando un paquete que le permite identificarse ante el servidor.

1. Pedidos

El cliente inicia la conexión enviando los datos de inicio de sesión, los campos son:

```
+-----+-----+-----+-----+-----+
| VER | ULEN |  USERNAME  | PLEN |  PASSWD  |
+-----+-----+-----+-----+-----+
|  1  |   1  | 1 to 255 |   1  | 1 to 255 |
+-----+-----+-----+-----+-----+
```

Dónde

- **VER** - Versión del protocolo
 - 0x01 Versión 1
- **ULEN** - Longitud del nombre de usuario
 - '0x00' - '0xFF' (0 - 255)
- **USERNAME** - Nombre de usuario
- **PLEN** - Longitud de la contraseña
 - '0x00' - '0xFF' (0 - 255)
- **PASSWD** - Contraseña

2. Respuestas

```
+-----+-----+
| VER | STATUS |
+-----+-----+
|  1  |   1   |
+-----+-----+
```

Dónde

- **VER** - Versión del protocolo
 - 0x01 Versión 1

- **STATUS** - estado de la conexión
 - 0x00 Conexión exitosa
 - 0x01 Versión no soportada
 - 0x02 usuario o password incorrecta
 - 0x03 fallo del servidor

En caso de que el status de la respuesta sea 0x01 o 0x02, la conexión se mantiene abierta para comenzar a enviar comandos o reintentar iniciar sesión según corresponda. En caso contrario, se cierra la conexión.

3. Ejemplo

pedido:

```
+-----+-----+-----+-----+
| VER | ULEN | UNAME | PLEN | PASSWD |
+-----+-----+-----+-----+
| 0x01 | 0x04 | 'juan' | 0x04 | '1234' |
+-----+-----+-----+-----+
```

Respuesta:

```
+-----+-----+
| VER | STATUS |
+-----+-----+
| 0x01 | 0x01 |
+-----+-----+
```

GET

1. Pedidos

```
+-----+-----+
| TYPE | CMD |
+-----+-----+
| 1 | 1 |
+-----+-----+
```

Dónde:

- **TYPE** se refiere al tipo de comando: '0x00' para GET
- **CMD** se refiere al comando:
 - '0x00' GET bytes transferidos
 - '0x01' GET conexiones históricas
 - '0x02' GET conexiones concurrentes
 - '0x03' GET listar usuarios

2. Respuestas

+	-----	+	-----	+	-----	+	-----	+
	STATUS		CMD		QARGS		ARGS	
+	-----	+	-----	+	-----	+	-----	+
	1		1		1		variable	
+	-----	+	-----	+	-----	+	-----	+

Dónde:

- **STATUS** se refiere al estado del pedido:
 - '0x00' Éxito
 - '0x01' Fallo del servidor
 - '0x02' Comando no soportado
 - '0x03' Tipo no soportado
- **CMD** se refiere al comando solicitado por el cliente:
 - '0x00' GET bytes transferidos (unsigned 32 bits big endian)
 - '0x01' GET conexiones históricas (unsigned 16 bits big endian)
 - '0x02' GET conexiones concurrentes (unsigned 16 bits big endian)
 - '0x03' GET listar usuarios (lista de usuarios codificados en ASCII de 7 bits)
- **QARGS** se refiere a la cantidad de argumentos que hay que parsear:
 - '0x00' - '0xFF' (0 - 255)
- **ARGS** se refiere a los argumentos del comando solicitado dónde el primer byte de cada argumento tiene la longitud del argumento.

3. Ejemplos

Pedido:

uint8_t pedido[] = {0x00, 0x01}

+	-----	+	-----	+
	TYPE		CMD	
+	-----	+	-----	+
	0x00		0x01	
+	-----	+	-----	+

Respuesta:

uint8_t respuesta[] = {0x00, 0x01, 0x01, 0x02, 0x27, 0xFB}

+	-----	+	-----	+	-----	+	-----	+
	STATUS		CMD		QARGS		ARGS	
+	-----	+	-----	+	-----	+	-----	+
	0x00		0x01		0x01		'10235'	
+	-----	+	-----	+	-----	+	-----	+

SET

1. Pedidos

+-----+	+-----+	+-----+	+-----+
TYPE	CMD	QARGS	ARGS
+-----+	+-----+	+-----+	+-----+
1	1	1	variable
+-----+	+-----+	+-----+	+-----+

Dónde:

- **TYPE** se refiere al tipo de comando: '0x01' para SET
- **CMD** se refiere al comando:
 - '0x00' SET agregar usuario/admin
 - Recibe **3** argumentos: tipo usuario y contraseña:
 - tipo: un byte que nos indica si se quiere agregar usuario o admin
 - '0x00': admin
 - '0x01': user
 - usuario: string ASCII, máx 255 caracteres
 - contraseña: string ASCII, máx 255 caracteres
 - '0x01' SET borrar usuario/admin
 - Recibe **1** argumento: usuario.
 - usuario: string ASCII, máx 255 caracteres
 - '0x02' SET cambiar contraseña de usuario/admin
 - Recibe **2** argumentos: usuario y nueva contraseña.
 - usuario: string ASCII, máx 255 caracteres
 - contraseña: string ASCII, máx 255 caracteres
 - '0x03' SET habilitar/deshabilitar el sniffer de contraseñas:
 - Recibe **1** argumento: prender o apagar
 - '0x00' prender (1 byte)
 - '0x01' apagar (1 byte)
 - '0x04' SET DoH IP: permite establecer la dirección del servidor DoH:
 - Recibe **2** argumento: el tipo de IP y la nueva dirección del servidor DoH
 - tipo de ip:
 - '0x00' = IPv4
 - '0x01' = IPv6
 - dirección: 4 o 16 bytes en big endian
 - '0x05' SET DoH Port: permite establecer el puerto del servidor DoH:
 - Recibe **1** argumento: el nuevo puerto del servidor DoH
 - puerto: 2 bytes en big endian
 - '0x06' SET DoH Host: permite establecer el valor del header host:
 - Recibe **1** argumento: el nuevo valor del header host.
 - host: string ASCII, máx 255 caracteres

- 0x07' SET DoH Path: permite establecer el path de la request DoH:
 - Recibe **1** argumento: el nuevo path del request DoH.
 - path: string ASCII, máx 255 caracteres
- 0x08' SET DoH Query: permite establecer el nuevo query string de la request DoH:
 - Recibe **1** argumento: el nuevo query string de la request DoH.
 - query: string ASCII, máx 255 caracteres
- **QARGS** se refiere a la cantidad de argumentos que hay que parsear:
 - '0x00' - '0xFF' (0 - 255)
- **ARGS** se refiere a los argumentos provistos dónde el primer byte de cada argumento tiene la longitud del argumento y cada argumento se codifica en ASCII de 7 bits

2. Respuestas

```

+-----+
| STATUS |
+-----+
|   1   |
+-----+

```

Dónde:

- **STATUS** se refiere al estado del pedido:
 - '0x00' Éxito
 - '0x01' Fallo del servidor
 - '0x02' Comando no soportado
 - '0x03' Tipo no soportado
 - '0x04' Error de argumentos
 - '0x05' Error no existe usuario
 - '0x06' Error no se soportan más usuarios/admins
 - '0x07' Error nombre de usuario tomado

3. Ejemplos

Un posible pedido sería el siguiente:

{0x01,0x00,0x03,0x01,0x01,0x04,0x68,0x6F,0x6C,0x61,0x03,0x31,0x32,0x033}

Dónde:

- TYPE: '0x01'
- CMD: '0x00'
- QARGS: '0x03'
- ARGS: [0x01,0x01,0x04,0x68,0x6F,0x6C,0x61,0x03,0x31,0x32,0x033]
 - Especificamos la longitud del primer argumento con el primer byte, luego especificamos que estamos queriendo agregar un admin con el '0x00', su nombre de usuario ocupa 4 bytes y se decodifica cómo

"hola", luego se especifica la longitud de la contraseña (3 bytes) y se decodifica cómo "123"

Una posible respuesta sería:

{0x00}

Dónde:

- STATUS: '0x00'

```
+-----+
| STATUS |
+-----+
|  0x00  |
+-----+
```

QUIT

Al ser orientado a sesión, tenemos que habilitar una manera de cortar la conexión. En una sesión abierta, enviando el primer byte con '0x02', se le indica al servidor que se quiere cerrar la conexión.

1. Pedido

```
+-----+
| TYPE  |
+-----+
|   1   |
+-----+
```

Dónde:

- **TYPE** se refiere al tipo de comando: '0x02' para QUIT

2. Respuesta

```
+-----+
| STATUS |
+-----+
|   1   |
+-----+
```

Dónde:

- **STATUS** se refiere al estado del pedido:
 - '0x00' Conexión cerrada exitosamente
 - '0x01' Fallo del servidor
 - '0x03' Tipo no soportado

3. Ejemplos

Un posible ejemplo de cierre de conexión sería el siguiente:

```
uint8_t peer[] = {0x02}
```

```
+-----+
| TYPE  |
+-----+
| 0x02  |
+-----+
```

Una posible respuesta sería:

```
uint8_t resp[] = {0x00}
```

```
+-----+
| STATUS |
+-----+
| 0x00  |
+-----+
```


Decisiones sobre el protocolo

Se decidió en la mayoría de los casos permitir argumentos variables debido a que habían peticiones que podían recibir cómo resultado una lista variable de respuestas, como en el caso de listar los usuarios. De la misma manera, tenemos comandos del tipo SET que envían distintas cantidades de argumentos y por lo tanto, con el fin de poder englobar todos los casos en un tipo de petición, se implementaron con argumentos variables. La manera en la que manejamos estos argumentos variables es que tenemos un byte que nos indica la cantidad de argumentos que se van a enviar y antes de cada argumento, tenemos un byte que nos indica la longitud de dicho elemento. El principal problema de esta implementación es que reduce el tamaño de los argumentos que se pueden enviar ya que antes de cada argumento debe ir su tamaño sólo se permiten 255 bytes para los argumentos.

Implementación en el proxy

Para implementar el protocolo en el proxy, se creo un socket SCTP, si bien este se puede configurar cómo SOCK_SEQPACKET, dónde el mensaje es enviado y recibido por el servidor completo, sin tener que esperar para recibir el paquete completo, se decidió utilizar la opción SOCK_STREAM, al igual que en TCP. Se tomó esta decisión ya que la primera opción no nos ofrecía grandes ventajas frente a la segunda, en particular, porque el mensaje recibido debía de todas formas parsearse byte a byte para responder correctamente al pedido. Adicionalmente, no pudimos crear un socket del protocolo para IPv4 y otro para IPv6 escuchando en el mismo puerto, ya que nos producía el siguiente error: “protocol not available”. Por esta razón, se utilizó un solo socket IPv4 escuchando por defecto en el puerto 8080.

El comando de bytes transferidos sólo cuenta los bytes transferidos del cliente al servidor de origen y viceversa. Tanto el comando de conexiones concurrentes cómo el de



conexiones históricas sólo cuentan conexiones por parte del cliente y sólo cuentan conexiones una vez que fueron establecidas con el origen. Estos datos junto con los otros que soporta el protocolo se almacenan en la misma estructura. Para la implementación de los usuarios, se utilizó una librería de tablas de hash (khash), manejamos 2 de estas tablas, una para usuarios comunes y la otra para administradores del sistema. Tanto los usuarios como los administradores pueden utilizarse para autenticarse con el protocolo SOCKSv5, sin embargo, sólo los administradores pueden utilizarse con el protocolo G8. Si bien ambos son guardados en distintas estructuras, los nombres de usuarios deben ser únicos en todo el sistema, es decir, no puede haber un admin y un usuario con el mismo nombre.

Problemas encontrados durante el diseño y la implementación

Los ejemplos que brinda el RFC8484 (que hace referencia a DoH) no nos parecieron muy explícitos, por lo que se nos dificultó comprender como se envía una request DoH y como se recibe una respuesta. Por eso, tuvimos que investigar en otras fuentes para unir la información y darle sentido a los conceptos del RFC. Pudimos sacarlo adelante, pero todo esto nos consumió más tiempo del que habíamos estipulado previamente.

Habíamos estipulado menos tiempo para comprender cuando era necesario setear los intereses para lectura y escritura del file descriptor. Tuvimos dificultades para hacerlo, y ello nos causó algunos leves retrasos.

Limitaciones de la aplicación

La principal limitación del proxy es que se ejecuta en un único hilo y, por tal motivo, la performance se degrada a medida que aumentan las conexiones concurrentes (soportando como máximo 1024). Esto es porque el selector es $O(N)$ al momento de iterar por los file descriptors que tienen operaciones por realizar, por lo que a mayor cantidad, más tiempo hay entre cada iteración.

Esto se podría mejorar paralelizando las conexiones en distintos hilos de ejecución y, de esta forma, poder atender múltiples conexiones al mismo tiempo.

Posibles extensiones

Con nuestra implementación, nos limitamos a implementar sólo 2 tipos de autenticación, sin autenticación y autenticación con usuario y contraseña, por lo que se podría implementar los métodos de autenticación especificados en el RFC 1928. De la misma manera, se podrían implementar los otros comandos especificados en el RFC 1928, ya que nuestra implementación sólo contempla el caso del 'connect'.

Adicionalmente, en la implementación de sniffers, se podría soportar pipelining de HTTP y POP3.

En relación a la resolución de nombres, nos dimos cuenta que también hay espacio para realizar mejoras. En primer lugar, se podría implementar pipelining, dándole la posibilidad al cliente de enviar varias requests juntas. En segundo lugar se podrían usar conexiones persistentes, y así reducir la cantidad de conexiones y mejorar el rendimiento del proxy.

Con respecto al protocolo que desarrollamos, si bien ahora el cliente puede dar uso de numerosos servicios, en el futuro se podrían agregar nuevas funcionalidades al mismo. En lo que concierne al servidor, una posible extensión sería soportar pipelining del protocolo, ya que al ser orientado a sesión sería se podría aprovechar bastante esta funcionalidad.

Conclusiones

A Pesar de los distintos inconvenientes que surgieron durante el diseño y desarrollo del trabajo, logramos obtener el resultado esperado: generar un funcionamiento correcto entre el servidor proxy que implementamos siguiendo las características detalladas en la introducción, y un cliente.

Apuntamos a hacer cada parte del trabajo de manera modularizada. De esa forma, pudimos lograr un avance constante y paralelo entre los diferentes integrantes del equipo y así evitar cuellos de botella. Luego, antes de integrar las diferentes partes, nos aseguramos de realizar los tests pertinentes para garantizar el correcto funcionamiento de las mismas.

Otra técnica que nos resultó de gran ayuda fue pensar y diagramar las máquinas de estado antes de implementar los parsers correspondientes. Esto nos permitió contemplar los diversos casos límite, que a veces en parsers tan largos son difíciles de notar.

A su vez, las herramientas y comandos de debugging como GDB y strace, fueron de gran ayuda para identificar más rápido errores que nos trabarían el desarrollo y así reducir tiempos de bloqueo significativos.

Creemos que este proyecto fue muy útil para consolidar todos los temas que vimos a lo largo del cuatrimestre y de esa manera construir un caso de uso real y pragmático. Sabemos que es un tema que tratamos con cotidianidad y aprender tanto del mismo nos enriqueció. Es por eso que nos interesa el hecho de que el trabajo se pueda extender en funcionalidades y utilidad. Creemos que contando con más tiempo, y partiendo sobre todo lo que construimos, podríamos tener algo aún más elaborado que genere un mayor impacto y agregue más valor a los usuarios finales.

Ejemplos de prueba

Se corrieron los siguientes tests:

En primer lugar, quisimos verificar la integridad de los datos que transportabamos, para esto se utilizó la herramienta curl para descargar una imagen ISO con y sin el proxy y lo pipepeamos a sha1sum para comprobar la integridad de la imagen ISO. Al mismo tiempo, también verificamos el tiempo que tardaba en descargar el paquete con y sin el proxy. Para esto se utilizaron imágenes de versiones viejas de Ubuntu de aproximadamente 500 MB, 1 GB y 2 GB. Se obtuvieron los siguientes resultados:

- Con 500 MB

Sin el proxy obtenemos los siguientes resultados:

```
> time curl http://old-releases.ubuntu.com/releases/4.10/warty-release-install-amd64.iso | sha1sum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100 540M 100 540M    0     0  9813k      0  0:00:56 0:00:56 --:--:-- 11.4M
79c4b34030ac55abe52a5c983b19452febbaa106 -
curl    0.86s user 6.81s system 13% cpu 56.386 total
sha1sum 4.61s user 1.56s system 10% cpu 56.386 total
```

Con el proxy y resolviendo la IP, obtenemos los siguientes resultados:

```
> time curl -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/4.10/warty-release-install-amd64.iso | s
ha1sum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100 540M 100 540M    0     0 10.1M      0  0:00:53 0:00:53 --:--:-- 10.4M
79c4b34030ac55abe52a5c983b19452febbaa106 -
curl -x socks5h://localhost:1080 0.94s user 5.31s system 11% cpu 53.389 total
sha1sum 3.63s user 0.88s system 8% cpu 53.389 total
```

Podemos observar que con archivos de menor tamaño, la velocidad

- Con 1 GB

Sin el proxy obtenemos los siguientes resultados:

```
> time curl http://old-releases.ubuntu.com/releases/14.04.0/ubuntu-14.04-desktop-amd64.iso | sha1sum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left     Speed
100 964M 100 964M    0     0 10.2M      0  0:01:33 0:01:33 --:--:-- 11.5M
d4d44272ee5f5bf887a9c85ad09ae957bc55f89d -
curl    2.02s user 12.26s system 15% cpu 1:33.90 total
sha1sum 8.04s user 3.31s system 12% cpu 1:33.90 total
```

Con el proxy y sin resolver la dirección IP, obtenemos los siguientes resultados:

```
> time curl -x socks5://localhost:1080 http://old-releases.ubuntu.com/releases/14.04.0/ubuntu-14.04-desktop-amd64.iso | shasum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left  Speed
100 964M 100 964M    0     0  8542k      0  0:01:55  0:01:55  --:--:-- 8595k
d4d44272ee5f5bf887a9c85ad09ae957bc55f89d -
curl -x socks5://localhost:1080  1.84s user 10.21s system 10% cpu 1:55.57 total
shasum  6.36s user 2.59s system 7% cpu 1:55.57 total
```

Con el proxy y resolviendo la dirección IP, obtenemos los siguientes resultados:

```
> time curl -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/14.04.0/ubuntu-14.04-desktop-amd64.iso | shasum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left  Speed
100 964M 100 964M    0     0 10.1M      0  0:01:35  0:01:35  --:--:-- 9752k
d4d44272ee5f5bf887a9c85ad09ae957bc55f89d -
curl -x socks5h://localhost:1080  1.63s user 10.51s system 12% cpu 1:35.35 total
shasum  6.61s user 2.11s system 9% cpu 1:35.35 total
```

- Con 2 GB

Sin el proxy obtenemos los siguientes resultados:

```
> time curl http://old-releases.ubuntu.com/releases/19.04/ubuntu-19.04-desktop-amd64.iso | shasum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left  Speed
100 2000M 100 2000M    0     0  8484k      0  0:04:01  0:04:01  --:--:-- 7807k
47064866141c7729b3f447890dd6d5bc2fc35cf7 -
curl  3.76s user 27.58s system 12% cpu 4:01.38 total
shasum 19.71s user 7.68s system 11% cpu 4:01.38 total
```

Con el proxy y resolviendo la IP, obtenemos los siguientes resultados:

```
> time curl -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/19.04/ubuntu-19.04-desktop-amd64.iso | shasum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left  Speed
100 2000M 100 2000M    0     0  9622k      0  0:03:32  0:03:32  --:--:-- 9853k
47064866141c7729b3f447890dd6d5bc2fc35cf7 -
curl -x socks5h://localhost:1080  3.70s user 23.24s system 12% cpu 3:32.85 total
shasum 14.57s user 4.42s system 8% cpu 3:32.85 total
```

En los 3 casos pudimos comprobar que la integridad de los archivos se mantuvo. En cuanto a la velocidad de descarga, vemos diferencias en los casos de 1 GB y 2 GB, pero vemos que se deben a la velocidad de descarga del archivo, si la velocidad de descarga fuese la misma el tiempo que toman sería el mismo.

Luego, se quiso verificar que con múltiples descargas simultáneas, nuevamente se mantenía la integridad de los datos. Se utilizó una imagen vieja de ubuntu de 1 GB y se compararon los sha1 resultantes, con y sin el proxy:

Sin el proxy se obtuvo el siguiente resultado:

```
scott@scott-VirtualBox:~$ curl http://old-releases.ubuntu.com/releases/14.04.0/ub
untu-14.04.1-desktop-amd64.iso | sha1sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100  981M  100  981M    0     0  8695k      0  0:01:55  0:01:55 --:--:-- 5554k
096dba6ee83d784009eddec7f28287ab66091f66 -
scott@scott-VirtualBox:~$
```

Con 4 conexiones simultáneas, se obtuvieron los siguientes resultados:

```
scott@scott-VirtualBox:~$ curl -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/14.04.0/ub
untu-14.04.1-desktop-amd64.iso | sha1sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100  981M  100  981M    0     0  2401k      0  0:06:58  0:06:58 --:--:-- 3866k
096dba6ee83d784009eddec7f28287ab66091f66 -
scott@scott-VirtualBox:~$

scott@scott-VirtualBox:~$ curl -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/14.04.0/ub
untu-14.04.1-desktop-amd64.iso | sha1sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100  981M  100  981M    0     0  2326k      0  0:07:11  0:07:11 --:--:-- 8915k
096dba6ee83d784009eddec7f28287ab66091f66 -
scott@scott-VirtualBox:~$

scott@scott-VirtualBox:~$ curl -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/14.04.0/ub
untu-14.04.1-desktop-amd64.iso | sha1sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100  981M  100  981M    0     0  2388k      0  0:07:00  0:07:00 --:--:-- 5594k
096dba6ee83d784009eddec7f28287ab66091f66 -
scott@scott-VirtualBox:~$

scott@scott-VirtualBox:~$ curl -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/14.04.0/ub
untu-14.04.1-desktop-amd64.iso | sha1sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100  981M  100  981M    0     0  2426k      0  0:06:54  0:06:54 --:--:-- 2160k
096dba6ee83d784009eddec7f28287ab66091f66 -
scott@scott-VirtualBox:~$
```

Por lo que podemos ver la integridad de los datos se mantuvo en las 4 conexiones.

Finalmente, para terminar de verificar que se mantenía la integridad de los datos, se hizo una descarga de un iso pasando por el proxy, cortado la descarga en el medio y retomándola inmediatamente después. Al resultado se le calculó el sha1 y se comparó con la misma descarga sin pasar por el proxy.

Sin pasar por el proxy:

```
scott@scott-VirtualBox:~$ curl http://old-releases.ubuntu.com/releases/4.10/warty-release-install-amd64.iso | sha1sum
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left    Speed
100 540M 100 540M    0     0  6698k      0  0:01:22  0:01:22 --:--:-- 7713k
79c4b34030ac55abe52a5c983b19452febaaa106 -
```

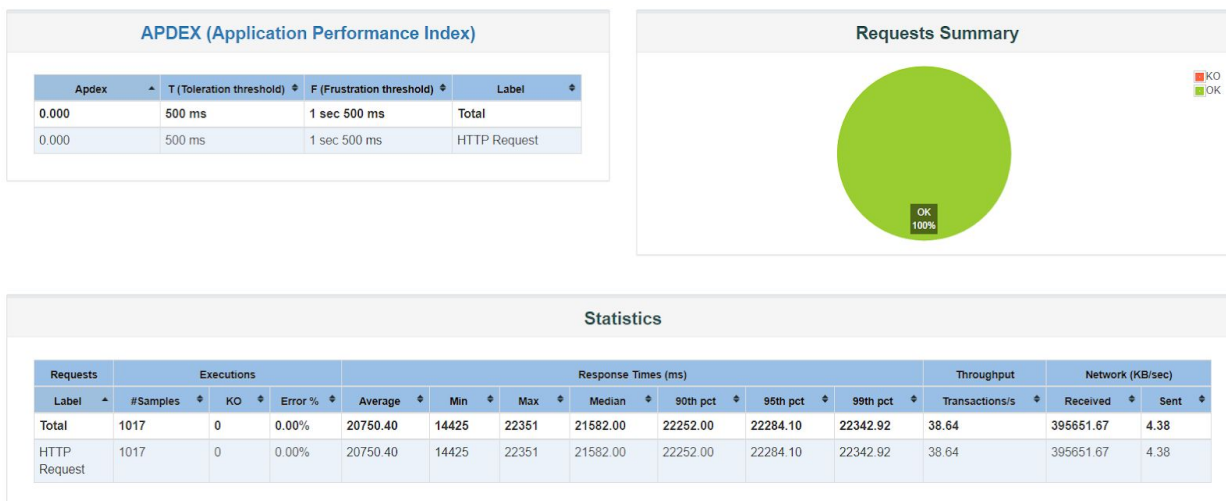
Pasando por el proxy y cortando al descarga:

```
scott@scott-VirtualBox:~$ curl --output ubuntun4-10.iso -C - -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/4.10/warty-release-install-amd64.iso
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left    Speed
58 540M 58 317M    0     0  7025k      0  0:01:18  0:00:46  0:00:32 8237k^C
scott@scott-VirtualBox:~$ curl --output ubuntun4-10.iso -C - -x socks5h://localhost:1080 http://old-releases.ubuntu.com/releases/4.10/warty-release-install-amd64.iso
** Resuming transfer from byte position 338460672
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left    Speed
100 217M 100 217M    0     0  6698k      0  0:00:33  0:00:33 --:--:-- 4005k
scott@scott-VirtualBox:~$ sha1sum ubuntun4-10.iso
79c4b34030ac55abe52a5c983b19452febaaa106 ubuntun4-10.iso
```

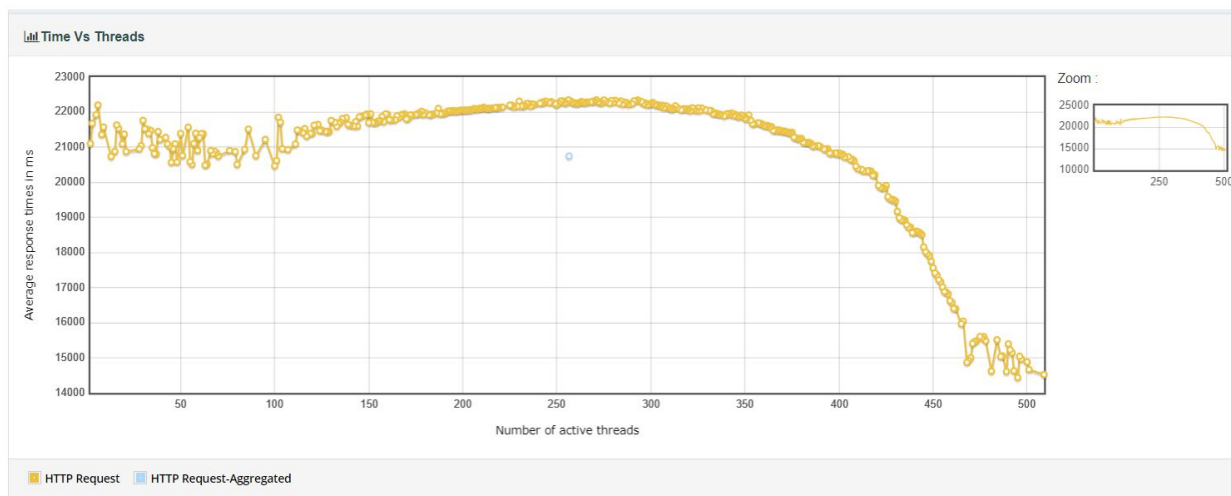
Cómo podemos ver el sha1 resultante es el mismo en ambos casos.

Con respecto a la prueba que realizamos para testear la cantidad de conexiones, se realizó una prueba para comprobar que soportamos al menos 500 clientes concurrentes. La prueba consistió en utilizar 509 threads, tiempo de ramp-up de 6 segundos y un solo loop descargando un archivo de 10 MB. Con esto no solo quisimos comprobar que soportamos los 509 clientes, ya que el máximo de file descriptors para un programa es de 1024 y tenemos 3 sockets pasivos, 1 file descriptor para STDOUT y 1 para STDERR. Esto nos deja con 1019 disponibles, y cómo para cada cliente hay que conectarlo con 1 servidor de origen, eso nos deja con un máximo de 509 file descriptors para clientes. Por lo tanto lo que esperamos observar en la prueba de 509 threads es que no tengamos errores.

Se obtuvieron los siguientes resultados en la prueba:



Como podemos ver en los resultados de la misma, todos los pedidos fueron exitosos, como se esperaba.



En cuanto al tiempo de respuesta, en el gráfico se puede observar que desde el comienzo hasta la mitad del test aproximadamente hay un leve incremento y luego comienza a bajar. Esto se debe a que hay una gran cantidad de conexiones al principio requiriendo información pero a medida que se van cerrando, el tiempo de respuesta para las conexiones que quedan va disminuyendo.

Guía de instalación detallada y precisa

Las instrucciones pertinentes se encuentran en el archivo README del proyecto, en la rama master.

Instrucciones para la configuración

A través de una aplicación que hemos desarrollado, el server otorga al cliente la posibilidad de realizar ciertas configuraciones, como sean cambiar el puerto, host, path o query del servidor. También, desde la misma, el cliente puede solicitar métricas sobre el uso del proxy. Para ver en mayor detalle las instrucciones de uso, recurrir al archivo README del proyecto, en la rama master.

Ejemplos de configuración y monitoreo

Ciente

Para utilizar el cliente del protocolo G8, es necesario tener un administrador registrado en el proxy, en caso de no tenerlo se puede usar el administrador por defecto, admin:admin. Para conectarse es necesario identificarse como administrador del proxy. Esto se logra enviando un flag -a user:pass con las credenciales o, en caso de ya existir, se le pedirán las credenciales para conectarse con el servidor.

Al conectarse al servidor, se muestra el siguiente menú:

```
01. (GET) transefered bytes
02. (GET) historical connections
03. (GET) concurrent connections
04. (GET) users list

05. (SET) add new users
06. (SET) remove user
07. (SET) change password to an user
08. (SET) enable/disable password sniffer
09. (SET) DOH IP
10. (SET) DOH port
11. (SET) DOH host
12. (SET) DOH path
13. (SET) DOH query
14. QUIT

Choose an option:
```

Aquí el usuario elige el comando que desea ejecutar introduciendo el número que identifica a la instrucción.

En base a la opción que seleccionó el usuario en el paso previo, a continuación mostramos dos ejemplos de la salida que se obtiene.

```
Choose an option: 1
total transfered = 4344430 B

Press [ENTER] to return
```

```
Choose an option: 2
Historical connections = 38

Press [ENTER] to return
```

Ciertas opciones opciones que requieren parámetros los pedirán al elegir el comando. Algunos comando también pedirán una confirmación adicional sobre el uso del comando:

```
Choose an option: 5
Adding new user

Username: juan

Password: 1234


Are you sure you want to perform this action? [Y]es or [N]o
y

Is this new user an Administrator? [Y]es or [N]o
y
Server response was: Success

Press [ENTER] to return
```


En caso de que el servidor responda con algún código de error, este será traducido correspondientemente. Por ejemplo, en caso de querer agregar un usuario o admin con un nombre ya existente, se retorna el error "User already exists", como se ve a continuación:

```
Choose an option: 5
Adding new user

Username: juan

Password: 777

Are you sure you want to perform this action? [Y]es or [N]o
y

Is this new user an Administrator? [Y]es or [N]o
n
Server response was: User already exists

Press [ENTER] to return
```

En este caso estamos removiendo un usuario.

```
Choose an option: 6
Removing an user

Username: juan

Are you sure you want to remove this user? [Y]es or [N]o
y
Server response was: Success

Press [ENTER] to return
```

En este caso se muestra el caso en el que se cambia la IP del servidor DoH, la cual puede ser tanto IPv4 como IPv6.

```
Choose an option: 9
Insert new IP address: 172.19.42.67
Are you sure you want to perform this action? [Y]es or [N]o
Y
Server response was: Success
Press [ENTER] to return
```

Diagrama de diseño del proyecto

