TRABAJO PRÁCTICO ESPECIAL
PROTOCOLOS DE COMUNICACIÓN

SERVIDOR PROXY
PROTOCOLO SOCKSv5

Dell'Isola, Lucas - 58025
Torrusio, Lucia - 59489

Descripción Servidor Proxy

La aplicación desarrollada en este trabajo práctico es un servidor Proxy SOCKS5, este puede atender múltiples conexiones concurrentes con operaciones no bloqueantes. El mismo permite autenticación con el cliente mediante usuario y contraseña, como se explica en el RFC 1929.

De los posibles métodos para autenticarse, nuestra aplicación soporta:

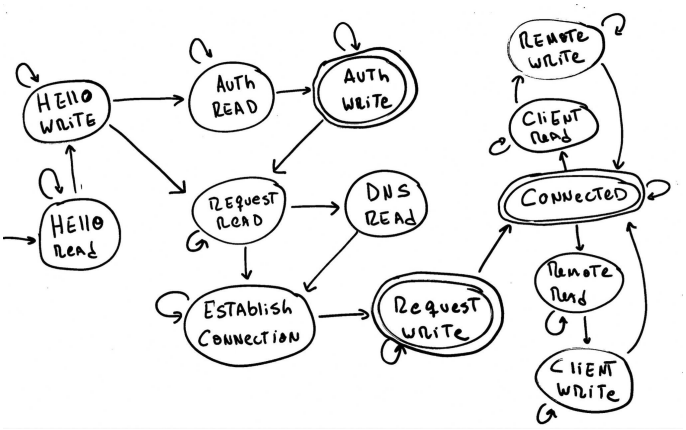
- Sin autenticación
- Autenticación con usuario y contraseña

  En el caso de conexión sin autenticación se le asigna un usuario anónimo “unknown”.

Esta implementación solo soporta el comando CONNECT, de intentarse conectar usando cualquier otro comando responderá de con la respuesta correspondiente (command not supported).

Flujo de la aplicación

Luego de inicializar todos los servicios requeridos, como lo son el servidor TCP, nuestra implementación del protocolo Socks5 y nuestro protocolo de management, cada cada conexión puede ser representada con la siguiente máquina de estados finita:



Donde todos los estados pueden llevar a una terminación temprana debido a un error de IO o desconexión.

Registro de accesos

Cuando una conexión cliente-servidor es establecida, el proxy lo informa imprimiendo en la pantalla el registro de acceso. Un posible ejemplo es el siguiente:

2022-06-21T03:29:33z	unknown	A	172.17.0.1	40833	www.google.com	80	0

En donde se informa:

- Instante de establecimiento de conexión
- Las credenciales de autenticación del cliente
- Tipo de registro
- IP del cliente
- Puerto de cliente
- IPv4/IPv6/FQDN del servidor remoto.
- Puerto destino.
- Estado de conexión

Registro de credenciales

Además de llevar un registro de acceso, el servidor tiene la funcionalidad de poder robar contraseñas que se envíen mediante el protocolo POP3. Estas serán impresas con el siguiente formato:

```
2022-06-21T09:54:06z    unknown    P    POP3    mail.btopenworld.com    110    lucas    secret
```

Donde los parámetros son:

- Instante de establecimiento de conexión
- Las credenciales de autenticación del cliente
- Tipo de registro
- IPv4/IPv6/FQDN del servidor remoto
- Puerto destino
- Nombre de usuario POP3
- Contraseña POP3

DNS

Otro detalle a tener en cuenta es que nuestro servidor no solo acepta conexiones IPv4 e IPv6, sino que también nos permite utilizar un FQDN, el cual será resuelto a un set de direcciones IP y se conectará a la primera disponible.

Para esto último se siguieron los lineamientos especificados en el RFC 8484.

De no poder conectarse a la primera dirección IP, va a seguir intentando conectarse al resto de las direcciones obtenidas a través del DNS. Si ninguna conexión es válida, entonces se le va a comunicar al cliente que la dirección no es válida con la respuesta apropiada.

Descripción de protocolo LULU

El protocolo LULU tiene como objetivo monitorear y configurar el servidor. Para su implementación se creó un protocolo propio el cual se encuentra detallado en el documento *Lulu Protocol.pdf.* Este protocolo funciona sobre TCP, y es de texto. Recibe los mensajes en el puerto 8080.

Nuestro protocolo LULU establece cómo se debe establecer la comunicación entre un cliente y un servidor proxy para que el primero pueda configurar y monitorear el funcionamiento del segundo. Este protocolo viaja sobre el protocolo de transporte TCP.
Este es utilizado para obtener métricas:

- Conexiones históricas.
- Cantidad de conexiones concurrentes
- Cantidad de bytes transmitidos

Es utilizado para obtener y cambiar valores de configuración:

- Timeout utilizado
- Tamaño de buffer

Y también para operaciones sobre los usuarios:

- Listar usuarios logueados.
- Crear un usuario
- Borrar un usuario

Se implementó un server LULU que atiende las conexiones de nuestro protocolo dentro de nuestra aplicación. Este cuenta con dos estados principales, autenticación y transacción. Necesariamente para poder acceder a los comandos de configuración y monitoreo se requiere una autenticación, utilizando usuario y contraseña. Luego de la autenticación se pasa al estado de transacción en el que se reciben los comandos mencionados anteriormente y se envía una respuesta.

También se implementó un cliente de LULU, el cual se ejecuta por línea de comando. Este cliente se conecta al server, realiza la autenticación, envía el pedido y luego devuelve en pantalla la respuesta que recibe de parte del server.

Para más detalles sobre este protocolo, se puede ver el RFC adjuntado dentro del repositorio.

Problemas encontrados durante el desarrollo

Uno de los problemas más grandes que encontramos a la hora de armar el trabajo práctico no fue el código en sí, sino las condiciones sobre las cuales estábamos trabajando. Debido a que los miembros del grupo no tienen los mismos espacios de desarrollo (Mac y Windows), decidimos que lo mejor sería programar utilizando un container de Docker y dejar que CLion se encargue de manejarlo.

Esto nos trajo dos problemas:

1. Es imposible correr el servidor y el cliente al mismo tiempo en modo debug. Esto nos obligó a programar nuestras implementaciones del cliente y servidor de forma completamente separada, utilizando netcat para simular a la parte faltante.
   Por un lado, nos obliga a tratar a nuestra definición del protocolo como un RFC completamente separado de nuestra implementación, pero también hace que los bugs sean más complicados de encontrar.
   Además, nos obligó a inicializar un container nuevo cada vez que queríamos correr de vuelta al programa, haciéndonos perder tiempo.

2. Estábamos al tanto de que en Pampero no se podían realizar las pruebas con IPv6. Pero, a pesar de que contamos con IPv6 en nuestras redes locales, el container solo tenía acceso a IPs versión 4. Este detalle nos costó bastante tiempo durante las etapas iniciales del desarrollo.

En cuanto al cliente, no encontramos ningún problema fundamentalmente complejo, sino que la falta de tiempo afectó severamente a la calidad de código y los logs. Esto nos causó otra gran pérdida de tiempo tratando de solucionar errores que, de haber implementado el cliente con más tiempo, podríamos haber evitado.

Limitaciones de la aplicación

La primera limitación de nuestra aplicación es la cantidad de usuarios concurrentes permitidos. Debido a nuestra decisión de utilizar el selector indicado por la cátedra, que depende de la función `pselect`, nuestro servidor está limitado a 1024 conexiones concurrentes, tomando en cuenta:

- Las conexiones Socks5
- Las conexiones LULU
- Los file descriptors utilizados a la hora de registrar los servicios LULU y Socks5.

La siguiente limitación viene por parte de nuestra implementación del servidor TCP. Nuestra implementación permite que haya un máximo de 50 conexiones en el backlog esperando a ser aceptadas por las interfaces de Socks5 o de LULU.

SOCKS5

En cuanto al servidor Socks5, tiene una limitación fundamental: solo acepta requests con el comando CONNECT. Esta decisión fue tomada ya que no era obligatorio implementar los otros comandos y preferimos utilizar ese tiempo para mejorar nuestra implementación.

Posibles extensiones

SOCKS5

Una clara posible extensión a nuestra implementación de SOCKS5 sería la implementación de los comandos BIND y UDP ASSOCIATE, para tener una versión completamente funcional del protocolo.

También se podrían agregar nuevas métricas generales, como la máxima cantidad de usuarios concurrentes, o hasta llegar a implementar métricas individuales para cada conexión, como la velocidad promedio del proxy.

Para mejorar la velocidad de nuestra implementación, podríamos tener una caché de los resultados del DNS, para evitar tener que hacer una llamada cada vez que tenemos que conectar a un dominio.

Por último, se podría seguir optimizando nuestra aplicación para eliminar la iteración sobre buffers innecesarios. Esto es algo que vimos en nuestras pruebas para investigar el tamaño apropiado de buffer.

LULU

Tanto para el cliente como para el protocolo, las posibles extensiones son:

- Agregar un sistema de versionado al protocolo.
- Actualizar el protocolo para interpretar las métricas propuestas anteriormente.

Y específicamente para el cliente, se podría implementar con una arquitectura simplificada que reuse código, ya que la implementación actual tiene bastante código repetido.

Conclusiones

Se logró crear un servicio proxy SOCKS5 y un funcionamiento correcto del cliente realizado acorde a lo que se esperaba de los mismos.

Durante el trabajo de desarrollo, nos fueron muy útiles los tests unitarios, ya que nos ayudaron a detectar varios errores en los parsers antes de que causen problemas. Si bien es una herramienta que habíamos utilizado en otros lenguajes, nunca lo habíamos implementado en C y la experiencia fue muy buena.

Por último, el uso de sockets no bloqueantes es una forma nueva de pensar al desarrollo de servidores en C, y no hubiese sido nuestra primera opción inicialmente. Habiendo terminado el desarrollo, nos pareció una gran herramienta y sin duda la consideraremos en otros proyectos.

Pruebas

Obviando las pruebas necesarias para validar nuestra implementación, como lo son verificar nuestro manejo de memoria con valgrind, decidimos hacer dos tipos de pruebas:

- Prueba de Estrés
- Prueba de Tiempo de Descarga

Prueba de Estrés

Nos interesaba bastante ver como nuestra implementación puede soportar grandes cantidades de conexiones en simultáneo, por eso decidimos utilizar stress tests.

Inicialmente pensábamos utilizar JMeter, ya que tiene muy buenas medidas sobre la degradación de la conexión, pero luego de intentar de configurarlo por varios días, decidimos darnos por vencidos.

Nuestra próxima idea fue utilizar K6, un framework de javascript que permite también realizar pruebas de estrés. Esta herramienta no nos fue útil ya que no soporta el uso de proxies.

Varios días después decidimos implementar nuestra propia herramienta para poder testear esta parte tan importante de nuestro servidor. En la entrega presentamos una versión compilada de este programa, así como el código fuente. Toda la información pertinente a esta aplicación se puede encontrar en su propio README.

Entendemos que nuestra implementación tiene limitaciones severas con respecto a los productos comerciales que intentamos utilizar, pero con los resultados que obtenemos podemos decir si la conexión se degrada o no.

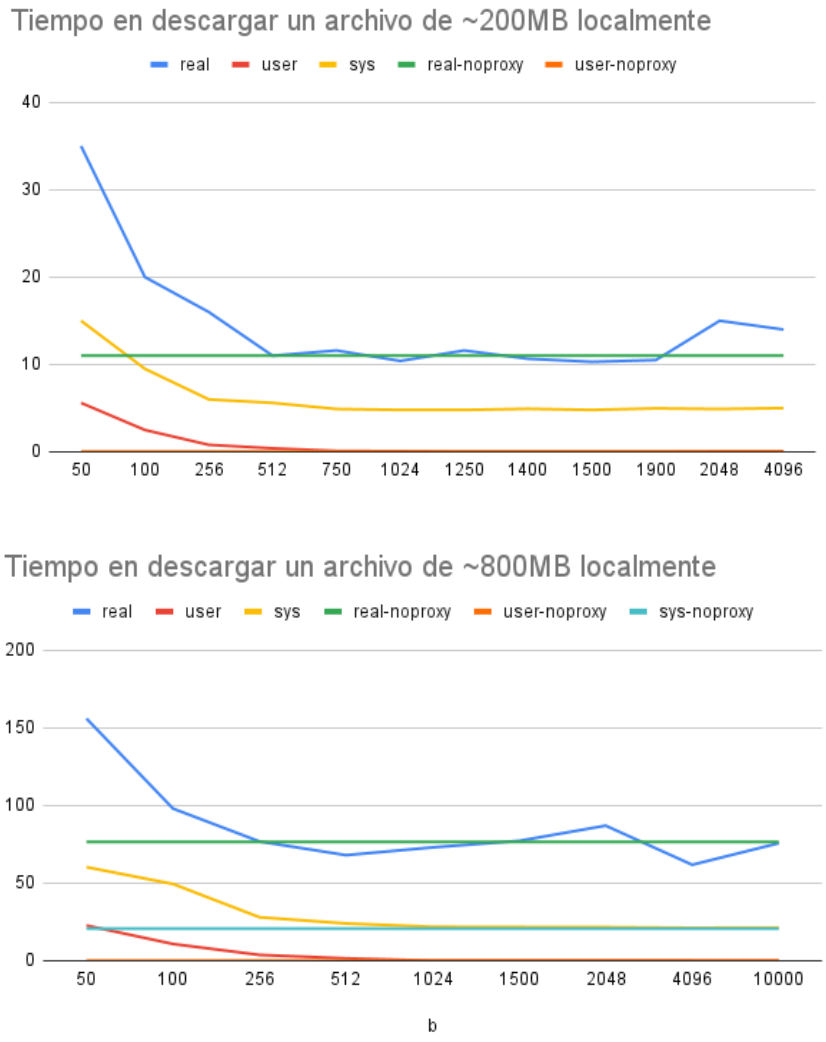
Un ejemplo de esto sería correr al test con 10 usuarios concurrentes, 3 requests por segundo y por un tiempo total de 60 segundos. El programa nos devuelve la cantidad de errores, junto a la cantidad total de requests que fueron realizados.

Con estos datos podemos obtener la cantidad de requests por segundo que se realizaron en realidad.

Pruebas de Tiempo de Descarga

Para probar el efecto del tamaño del buffer, decidimos levantar un servidor http local con dos archivos, uno mediano de 200 MB y uno grande de 800 MB (los tamaños son una limitación del almacenamiento máximo que tenemos en pampero).

En los gráficos podemos admirar como el tiempo de descarga es afectado por los distintos tamaños de buffer:



Luego de realizar estos tests, definimos a nuestro tamaño de buffer como 1024, ya que lo consideramos como un buen balance entre el tamaño que ocupa en memoria por cada conexión y la cantidad de veces que tiene que leer del buffer por cada request.

Guía de instalación

Para compilar tanto el cliente como el servidor es necesario tener instalado `CMAKE`:

```
$ apt-get install cmake
```

Para poder compilar el programa hay que seguir los siguientes pasos:

1. Clonar el repositorio y entró a la carpeta:
2. `$ git clone https://github.com/ldellisola/socks5d.git`
3. `$ cd socks5d`
4. Entró a la carpeta del código fuente e inicializo el proyecto CMake
5. `$ cd socks5d`
6. `$ cmake .`
7. Ahora puedo compilar ambos programas:
    - Para compilar el servidor:
    - `$ cd src/server`
    - `$ cmake --build .`
    - Para compilar el cliente:
    - `$ cd src/lulu_client`
    - `$ cmake --build .`

Instrucciones de configuración

Las instrucciones de configuración tanto del servidor como del cliente están almacenadas en el repositorio, representadas en la forma de un archivo man. Estos están en la carpeta *references*.

En cuanto a la implementación del servidor, agregamos 2 parámetros extra y una variable de entorno:

- **-b** indica el tamaño de buffer a utilizar. Por default tiene el valor 1024.
- **-t** indica el timeout a utilizar. De ser 0 o negativo se deshabilita el timeout. Por default tiene 500 segundos
- La variable de entorno `SOCKS5D_LOG_LEVEL` indica el nivel de log, que por defecto está en `INFO`. Se pueden utilizar los siguientes valores:
    - `DEBUF`
    - `INFO`
    - `WARN`
    - `ERROR`
    - `FATAL`

En cuanto al cliente, este tiene los siguientes parámetros:

- **-u** Indica el usuario para conectarse al servidor LULU. Este parámetro es obligatorio y tiene que ir en la forma “<user>:<pass>”.
- **-P** Indica el puerto a ser utilizado
- **-L** Indica la IP a la cual conectarse.

También se puede utilizar la misma variable de entorno para configurar el nivel de log.

Ejemplos

Nuestra implementación del cliente puede ser ejecutada de la siguiente manera:

```
$ ./client -u user:pass -L 127.0.0.1 -P 1234
```

Para conectarse a la dirección 127.0.0.1, en el puerto 1234 como el usuario user con la contraseña pass.

El servidor puede ser ejecutado de la siguiente manera:

```
$ ./socks5d -u sUser:sPass -l ::1 -p 9000
```

Para ejecutar al cliente en la IPv6 ::1, sobre el puerto 9000 con el usuario sUser y la contraseña sPass.