TRABAJO PRÁCTICO ESPECIAL

PROTOCOLOS DE COMUNICACIÓN

SERVIDOR PROXY
PROTOCOLO SOCKSv5

insertar legajos

Barbieri, Guido - 59567
Brula, Matías Alejandro - 58639
Tallar, Julián - 59356
Vuoso, Julián Matías - 59347

Índice

| Introducción **4**

| Protocolo de configuración y monitoreo sobre SCTP **5**

| Aplicaciones desarrolladas **14**

Servidor proxy 14
Usuarios 14
Funcionamiento general del proxy 15
Resolución de nombres 15
Conexión 16
Manejo de errores 17
Intercambio de data 17
Disector de credenciales 17
Monitoreo y configuración 18
Cliente del protocolo 19

| Problemas encontrados **19**

| Limitaciones de la aplicación **20**

| Posibles extensiones **21**

| Conclusiones **21**

| Ejemplos de prueba **22**

| Guía de instalación **22**

| Instrucciones para la configuración **23**

| Ejemplo de configuración y monitoreo **23**

| Documento de diseño del proyecto (Arquitectura de la aplicación) **23**

| Introducción

El objetivo de este trabajo práctico es implementar un servidor proxy para el protocolo SOCKS v5 detallado en el *RFC1928*. El servidor debe dar soporte para múltiples clientes en forma concurrente, quienes deben autenticarse con un usuario y contraseña como indica el *RFC1929*. También debe poder resolver requests a direcciones IPv4, IPv6 o FQDN, resolviendo estos últimos nombres con DNS sobre HTTP, como indica el *RFC8484*. Además, debe mantener un registro de acceso para saber quién se conectó, a qué sitio web y cuándo, y debe poder monitorear el tráfico para registrar credenciales de acceso para HTTP y POP3. Por último, debe contar con un mecanismo vía SCTP para monitorear y configurar el servidor.

| Protocolo de configuración y monitoreo sobre SCTP

El siguiente protocolo establece cómo DEBE ser 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 SCTP.

Autenticación del cliente

El cliente se conecta al servidor y envía un REQUEST con la versión del protocolo, su nombre de usuario y contraseña para acceder a las opciones de configuración y monitoreo del servidor.

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

Donde:

- VER -> versión actual del protocolo, que es X'01'.
- ULEN -> largo del campo UNAME que sigue
- UNAME -> nombre de usuario
- PLEN -> largo del campo PASSWD que sigue
- PASSWD -> contraseña asociada al UNAME dado

El servidor verifica el nombre de usuario y contraseña suministrados y envía la siguiente respuesta

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

Donde:

- VER -> versión actual del protocolo, que es X'01'.
- STATUS -> estado de respuesta

Un campo STATUS con X'00' indica éxito. Si el servidor retorna falla (cualquier valor distinto de X'00') el cliente DEBE cerrar la conexión.

REQUESTS

Una vez completado el proceso de autenticación, el cliente procede a enviar los comandos que desee de los que se listan a continuación.
A cada comando le corresponde un formato de REPLY particular, especificado a continuación de la explicación del comando. Todo número que ocupe más de 1 byte debe ser interpretado como un número en "network octet order".
El cliente PUEDE enviar un comando seguido de otro, a lo cual el servidor DEBE responder en orden.

Los códigos de CMD existentes son

~ ADD USER	X'00'
~ DELETE USER	X'01'
~ LIST USERS	X'02'
~ GET METRIC	X'03'
~ GET CONFIG	X'04'
~ SET CONFIG	X'05'

Los códigos de STATUS existentes son

~ SUCCESS	X'00'
~ INV CMD	X'01'
~ INV ULEN	X'02'
~ INV UTYPE	X'03'
~ INV PLEN	X'04'
~ INV METRIC	X'05'
~ INV CONFIG	X'06'
~ MAX UCOUNT	X'A0'
~ INV VALUE	X'A1'
~ GEN SERV FAIL	X'FF'

En cualquiera de los comandos, el servidor debe enviar un STATUS de GEN SERV FAIL en caso de que haya ocurrido un error irrecuperable de funcionamiento en el mismo, sin relación con lo enviado por el cliente.
Si el cliente envía un valor inválido, el servidor DEBE cerrar la conexión SCTP con el cliente en un lapso de tiempo corto después de enviar la REPLY.
Si el cliente envía un valor válido pero el servidor se encuentra limitado por su implementación (errores X'A0' a X'AF'), el cliente PUEDE volver a enviar un nuevo REQUEST luego de leer la respuesta.

Agregar Usuario

CMD	UTYPE	ULEN	UNAME	PLEN	PASSWD
1	1	1	1 to 255	1	1 to 255

Donde:

- CMD X'00'
- UTYPE -> tipo de usuario
  - ~ CLIENT X'00'
  - ~ ADMIN X'01'
- ULEN -> largo del campo UNAME que sigue
- UNAME -> nombre de usuario
- PLEN -> largo del campo PASSWD que sigue
- PASSWD -> contraseña asociada al UNAME dado

El servidor verifica que el largo sea válido (mayor a 0) y que el tipo de usuario sea correcto, y si lo es, intenta agregar al usuario a la lista.
Si el nombre de usuario ya existía, se actualiza su contraseña y su tipo de usuario.
Luego envía la siguiente REPLY

CMD	STATUS
1	1

Donde:

- CMD X'00'
- STATUS -> estado de respuesta
  - ~ SUCCESS X'00'
  - ~ INV ULEN X'02'
  - ~ INV UTYPE X'03'
  - ~ INV PLEN X'04'
  - ~ MAX UCOUNT X'A0'
  - ~ GEN SERVER FAIL X'FF'

En caso de recibir INV ULEN, INV UTYPE, o INV PLEN el cliente NO PUEDE volver a enviar un nuevo REQUEST.
De recibir MAX UCOUNT , el cliente PUEDE volver a enviar un mensaje.

Eliminar usuario

CMD	ULEN	UNAME
1	1	1 to 255

Donde:

- CMD X'01'
- ULEN -> largo del campo UNAME que sigue
- UNAME -> nombre de usuario a remover

El servidor intenta eliminar al usuario a la lista. Si el nombre de usuario no existe, no hace nada y responde SUCCESS (es idempotente).
Luego envía la siguiente REPLY

CMD	STATUS
1	1

Donde:

- CMD X'01'
- STATUS -> estado de respuesta
  - ~ SUCCESS X'00'
  - ~ GEN SERVER FAIL X'FF'

Listar nombres de usuario

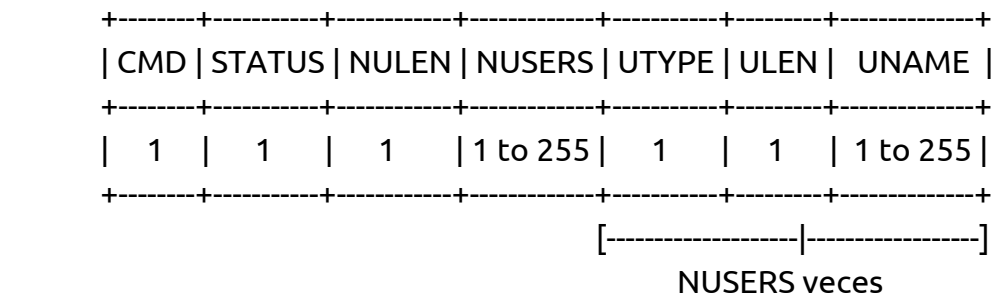
CMD
1

Donde:

- CMD X'02'

El servidor intenta listar todos los nombres de usuario existentes junto con su tipo.

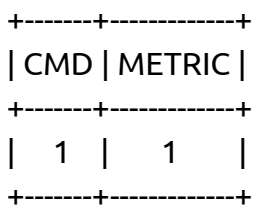
Luego envía la siguiente REPLY



Donde:

- CMD X'02'
- STATUS -> estado de respuesta
  - ~ SUCCESS X'00'
  - ~ GEN SERVER FAIL X'FF'
- NULEN -> (largo - 1) del campo NUSERS que sigue
- NUSERS -> cantidad de usuarios a continuación
- ULEN -> largo del campo UNAME que sigue
- UNAME -> nombre del usuario
- UTYPE -> tipo de usuario
  - ~ CLIENT X'00'
  - ~ ADMIN X'01'

Obtener métrica



Donde:

- CMD X'03'
- METRIC -> métrica solicitada
  - ~ HISTORIC CONNECTIONS X'00'
  - ~ CONCURRENT CONNECTIONS X'01'
  - ~ HISTORIC BYTES TRANSFERRED X'02'

El servidor intenta obtener la métrica solicitada. Luego envía la siguiente REPLY

CMD	STATUS	METRIC	VLEN	VALUE
1	1	1	1	0 to 255

Donde:

- CMD X'03'
- STATUS -> estado de respuesta
  - ~ SUCCESS X'00'
  - ~ INV METRIC X'05'
  - ~ GEN SERVER FAIL X'FF'
- METRIC -> métrica solicitada
  - ~ HISTORIC CONNECTIONS X'00'
  - ~ CONCURRENT CONNECTIONS X'01'
  - ~ HISTORIC BYTES TRANSFERRED X'02'
- VLEN -> largo del campo VALUE que sigue
- VALUE -> valor de la métrica solicitada

El valor obtenido debe ser interpretado como un número, que estará en la unidad Bytes en caso de que la configuración solicitada sea X'02'.

En caso de recibir INV METRIC, el cliente PUEDE leer los tres campos siguientes según el valor de VLEN, aunque NO PODRÁ volver a enviar un nuevo REQUEST.

Obtener configuración

CMD	CONFIG
1	1

Donde:

- CMD X'04'
- CONFIG -> configuración solicitada
  - ~ BUFFER READ X'00'
  - ~ BUFFER WRITE X'01'
  - ~ GENERAL TIMEOUT X'02'
  - ~ CONNEC TIMEOUT X'03'

El servidor intenta obtener la configuración solicitada. Luego envía la siguiente REPLY

CMD	STATUS	CONFIG	VLEN	VALUE
1	1	1	1	0 to 255

Donde:

- CMD X'04'
- STATUS -> estado de respuesta
  - ~ SUCCESS X'00'
  - ~ INV CONFIG X'06'
  - ~ GEN SERVER FAIL X'FF'
- CONFIG -> configuración solicitada
  - ~ BUFFER READ X'00'
  - ~ BUFFER WRITE X'01'
  - ~ GENERAL TIMEOUT X'02'
  - ~ CONNEC TIMEOUT X'03'
- VLEN -> largo del campo VALUE que sigue
- VALUE -> valor de la configuración solicitada

En caso de ser un tamaño de buffer (configuraciones X'00', X'01', X'02'), el valor obtenido debe ser interpretado como un número en unidad bytes.
Mientras que si la configuración solicitada es de un timeout (configuración X'03'), el valor obtenido debe ser interpretado como un número en unidad segundos.

En caso de recibir INV CONFIG, el cliente PUEDE leer los tres campos siguientes según el valor de VLEN, aunque NO PODRÁ volver a enviar un nuevo REQUEST.

Cambiar configuración

CMD	CONFIG	VLEN	VALUE
1	1	1	0 to 255

Donde:

- CMD X'05'
- CONFIG -> configuración solicitada
  - ~ BUFFER READ X'00'
  - ~ BUFFER WRITE X'01'
  - ~ GENERAL TIMEOUT X'02'
  - ~ CONNEC TIMEOUT X'03'
- VLEN -> largo del campo VALUE que sigue
- VALUE -> valor de la configuración solicitada

En caso de ser un tamaño de buffer (configuraciones X'00', X'01', X'02'), el valor dado será interpretado como un número en unidad bytes.
Mientras que si la configuración solicitada es de un timeout (configuración X'03'), el valor dado será interpretado como un número en unidad segundos.

El servidor valida el valor dado y, de ser correcto, intenta aplicar el cambio de configuración. Luego envía la siguiente REPLY

CMD	STATUS	CONFIG	MLEN	MESSAGE
1	1	1	1	0 to 255

Donde:

- CMD X'05'
- STATUS -> estado de respuesta
  - ~ SUCCESS X'00'
  - ~ INV CONFIG X'06'
  - ~ INV VALUE X'A1'
  - ~ GEN SERVER FAIL X'FF'
- CONFIG -> configuración solicitada
  - ~ BUFFER READ X'00'
  - ~ BUFFER WRITE X'01'
  - ~ GENERAL TIMEOUT X'02'
  - ~ CONNEC TIMEOUT X'03'
- MLEN -> largo del campo MESSAGE que sigue
- MESSAGE -> mensaje que explica el error (de ser necesario)

En caso de recibir INV CONFIG, el cliente PUEDE leer los campos que siguen, aunque NO PODRÁ volver a enviar un nuevo REQUEST.

En el caso de recibir INV VALUE, el servidor PUEDE enviar una cadena de caracteres en MESSAGE, indicando su largo en MLEN, explicando por qué el valor es inválido. Por lo tanto el cliente DEBE leer los campos que le siguen. Luego PUEDE volver a enviar otro REQUEST.

Comando inválido

En caso de que el comando ingresado no corresponda con los valores listados anteriormente, el servidor envía la siguiente REPLY

CMD	STATUS
1	1

Donde:

- CMD -> el comando solicitado
- STATUS -> estado de respuesta
  - ~ INV CMD X'01'
  - ~ GEN SERVER FAIL X'FF'

En caso de recibir INV CMD, el cliente NO PODRÁ volver a enviar un nuevo REQUEST.

| Aplicaciones desarrolladas

Se desarrollaron dos aplicaciones: **socks5d** y **client**. La primera se corresponde con el servidor proxy para el protocolo SOCKS v5, mientras que la segunda consiste en una posible implementación de un cliente del protocolo desarrollado para operaciones de configuración y monitoreo del servidor.

Servidor proxy

El servidor abre y configura los sockets correspondientes para poder aceptar conexiones en la dirección y puerto especificados por argumentos, como indica el manual socks5d, tanto para atender conexiones de clientes como de administradores. Los clientes se conectarán mediante TCP, mientras que los administradores se conectarán vía SCTP.

La implementación no bloqueante del servidor se basa en la versión del selector provista por la cátedra, con algunas leves modificaciones. Luego de realizar toda su inicialización y registrar los file descriptors de los sockets anteriores con interés en su lectura, simplemente se llama al selector_select en un loop hasta interrumpir la ejecución con una señal de SIGINT o SIGTERM.

Además, aprovechando que podemos configurar un timeout en el selector, agregamos otros dos timeouts propios para que, pasado cierto tiempo de su último uso, se des-registre y se cierren los file descriptors inactivos o “trabados”. Esto se hizo agregando en cada ítem el tiempo de última actualización (registro, lectura, escritura o señal de desbloqueo) y comparando el tiempo actual con éste último, teniendo en cuenta el valor fijado de timeout. Hay que considerar que, por obvias razones, el timeout está deshabilitado para los descriptores de los sockets creados para aceptar nuevas conexiones.

Se establecen dos tipos de timeout: uno para conexiones en proceso y otro para conexiones establecidas. El primero se utiliza para poder interrumpir syscalls de connect que tardan mucho en determinar que no puede alcanzar a un host. Mientras que el segundo es para cerrar conexiones que hayan estado “idle” durante un largo lapso de tiempo.

Usuarios

Para el manejo de usuarios, al iniciar el servidor se intenta abrir un archivo users.txt ubicado en el mismo directorio en el que se ejecuta el servidor. En caso de existir, se inicializa la lista de usuarios con el contenido de este archivo. Se espera que los usuarios estén en el formato *username:password:level*, siendo level el nivel de acceso del usuario.

Como se explica en el manual, los niveles de acceso son cliente (nivel 0) y administrador (nivel 1). El cliente solamente puede acceder al proxy como tal,

mientras que el administrador puede acceder tanto para utilizarlo como cliente como para realizar tareas de configuración y monitoreo.

Para agregar credenciales sin el archivo mencionado, se puede ejecutar el servidor con los parámetros -u o -U (explicados en detalle en el manual) o se puede utilizar el cliente de nuestro protocolo con un usuario administrador para agregar usuarios en tiempo de ejecución.

Previo a finalizar la ejecución del servidor, se crea o actualiza el archivo users.txt (según exista previamente o no) con los valores de la lista de usuarios vigente.

Funcionamiento general del proxy

Para lograr el funcionamiento del proxy, se utiliza una máquina de estados basada en el pequeño motor de máquinas de estado provisto por la cátedra. En ella, tendremos cada uno de los estados generales por los que pasa el proxy, mientras que en cada estado tendremos (de ser necesario) un parser particular del mismo.

La máquina de estados permite configurar 5 eventos por cada estado: cuando se llega al mismo, cuando la lectura está lista, cuando la escritura está lista, cuando se puede resolver un evento bloqueante y cuando se sale del estado. De esta manera, cada vez que se produce un evento sobre un file descriptor simplemente se delega ese evento a la máquina de estados.

En cada uno de los parsers, analizamos byte a byte, de forma tal de no depender de la cantidad de bytes recibidos en cada ocasión. En el caso del estado de hello, solamente se aceptan clientes que ofrecen como método usuario/contraseña. Luego, en el estado de negociación, se verifica que exista el usuario indicado por el cliente (con cualquier nivel, ya sea cliente o administrador) y se comprueba que la contraseña sea correcta. En caso de serlo, el cliente procede a enviar el request.

En el procesamiento del request, al haber tres posibles caminos (IPv4, IPv6 o FQDN), se buscó generalizar la parte de conexión al servidor, completando en los 3 casos un puntero a una estructura addrinfo. De esta forma, podemos guardar ambas versiones del protocolo IP, así como manejar listas de direcciones en el caso de tener que resolver nombres. Solamente se aceptan aquellos requests con el comando CONNECT.

Resolución de nombres

Para resolver FQDNs, haremos consultas DNS sobre HTTP, como indica el *RFC8484*. El request HTTP podría ser mediante un GET, donde la consulta va por parámetros de la URL, o mediante POST, donde la consulta va en el cuerpo del request. Decidimos hacerlo vía GET ya que nos pareció más correcto dada su idempotencia, lo que permitiría por ejemplo cachear los requests.

Por otro lado, decidimos no manejar conexiones persistentes para mayor simpleza de la implementación. Por lo tanto, cuando se desea resolver un nombre seguimos los siguientes pasos:

1. Abrimos un socket y establecemos la conexión con el servidor de DNS sobre HTTP
2. Armamos la query DNS codificada en base64 en función al nombre a resolver
3. Enviamos el request GET HTTP al servidor consultando por los registros de tipo A, es decir, para obtener direcciones IPv4
4. Leemos la respuesta y la parseamos para obtener la lista de direcciones
5. Si el parseo fue exitoso, guardamos la lista de direcciones en la estructura addrinfo mencionada anteriormente y procedemos a la etapa de conexión.

En caso de no poder conectarnos a ninguna de esas direcciones, volvemos a repetir el procedimiento, pero esta vez consultando por registros de tipo AAAA para obtener direcciones IPv6.

Existe la posibilidad de que no se pueda establecer la conexión con ninguna de las direcciones obtenidas, o bien puede haber alguna falla previa, como no poder establecer la conexión con el servidor o algún error en la lectura o escritura de su file descriptor. En cualquiera de los dos casos, se procede como default a intentar resolver el nombre con la función getaddrinfo.

Como esta función es bloqueante y no queremos que ninguna función bloquee el funcionamiento del servidor, abrimos otro hilo, donde se ejecutará dicha función. Una vez finalizada la misma, se intenta establecer la conexión con alguna de las direcciones obtenidas.

Conexión

Una vez que tenemos una o más direcciones IP a las cuales intentar conectarnos, procedemos a la etapa de conexión. En ésta, se abre un socket y se intenta conectar a la dirección en cuestión. Hay que tener en cuenta que la syscall de connect puede llegar a devolver error, con código de error EINPROGRESS. En este caso, debemos suscribirnos a la escritura del file descriptor para, una vez que se pueda escribir en el mismo, leer la opción SO_ERROR para determinar si efectivamente se pudo establecer o no la conexión.

En caso de fallar la conexión con una dirección IP, se pasa a la siguiente en la lista hasta que podamos conectarnos o que lleguemos al final de la lista. Es aquí donde el timeout para conexiones en proceso cobra una vital importancia, ya que muchas veces la syscall tarda mucho tiempo en reconocer que no puede llegar a un host.

Manejo de errores

Luego de terminar la etapa de conexión a la dirección o el nombre solicitado, debemos informar al cliente cuál fue el resultado de su request, haya sido exitoso o no. Es por ello que, sin importar si se logró establecer la conexión o no, se procede a enviar una reply al request del cliente, con el código de respuesta fijado acordemente.

Siempre que el protocolo lo permita, se intenta ser lo más específico posible con el código de respuesta, cayendo en el código 0x01 de falla general del servidor SOCKs en caso de no ser posible. En la salida estándar, se registra la información del request con la fecha, el nombre de usuario, la IP y puerto de origen, el destino (en el formato recibido en el request) y el puerto destino, y el status de la conexión, según el *RFC1928*.

Intercambio de data

Una vez establecida la conexión, se procede al intercambio de datos entre el cliente y el servidor de origen al que deseaba conectarse. Inicialmente, el proxy está interesado en leer a cualquiera de los dos file descriptors, ya que, dependiendo del protocolo manejado por el cliente y el origen, será uno u otro el primero en enviar un mensaje.

Aquí es donde el proxy actúa como tal, enviando lo recibido desde el cliente al servidor de origen y viceversa. Lo importante es manejar correctamente los intereses (lectura y escritura) en cada extremo. Cuando el buffer de lectura de un extremo se llena, se deshabilita la lectura en esta punta, ya que no podremos guardar lo leído. Cuando leemos algo de un extremo, habilitamos la escritura en el otro ya que debemos copiar lo leído allí. Una vez que copiamos todo el contenido del buffer de escritura en un extremo, deshabilitamos la escritura porque ya no tenemos nada que escribir.

Cuando la syscall recv retorna 0 o retorna error de código ECONNRESET, el extremo cerró la conexión. Debemos terminar de enviar lo que estaba pendiente de envío y enviar un shutdown al otro extremo para indicarle que ya no va a recibir más datos. Este extremo terminará de enviar lo que le quedó pendiente y cerrará su conexión, lo que provocará lo mismo que antes, pero en el sentido inverso.

Disector de credenciales

En el estado de copia anterior, debemos también monitorear el tráfico para registrar credenciales de acceso en caso de que la comunicación sea vía HTTP o POP3. Para discernir entre ambos protocolos, tomamos como dato el puerto al que se estableció la conexión en el servidor de origen. Si se trata del puerto 110, se utiliza un parser para POP3. En otro caso, se intenta parsear para HTTP.

En caso de tratarse de una conexión HTTP, se espera que la conexión sea un request GET con algún path, versión HTTP/1.1, y se van parseando los headers buscando “authorization”. En caso de encontrarlo y que su tipo sea basic (base64), se leen las credenciales hasta encontrar un \r. Luego se decodifica el usuario y la contraseña, y se imprimen las credenciales en pantalla.

En caso de tratarse de una conexión POP3, se espera a leer las palabras “user” y “pass”, ignorando todas las líneas que no contengan dichas palabras. Una vez encontradas, se imprimen las credenciales en pantalla.

En cualquiera de los casos, se intentó no perjudicar la performance en gran manera. Por ello, solamente se lee la primera vez que se envían las credenciales, hayan fallado o no. Tampoco se realizaron chequeos en HTTP de que el path del request esté bien formado o que el request HTTP completo sea correcto. Una vez obtenidas las credenciales o que se determina que la información no es del interés del disector, se lo deshabilita y se procede únicamente a copiar datos de un extremo a otro en este estado.

Monitoreo y configuración

Para lograr el funcionamiento del monitoreo y configuración del proxy siguiendo el protocolo especificado previamente, realizamos una nueva máquina de estados similar a la máquina de estados de funcionamiento general del proxy. Basados nuevamente en el pequeño motor de máquinas de estado provisto por la cátedra, ésta vez tendremos solamente 5 casos.

Primero, se espera que el administrador se autentique tal como especifica el protocolo. Como mencionamos anteriormente, la lista de usuarios es la misma que la del proxy, solo que, para que el usuario sea autenticado, su nivel debe ser de administrador (nivel 1). En cualquiera de los dos casos, se responde con un mensaje indicando si la autenticación fue exitosa o no.

Una vez autenticado, se pasa a un estado de lectura y escritura de comandos. El cliente puede enviar comandos especificados en el protocolo y recibirá las respuestas a los mismos (en orden, si envía más de uno). Una vez que cierre su conexión, el servidor termina de enviar los mensajes que hayan quedado pendientes y cierra la conexión.

Algunos detalles particulares de la implementación del protocolo son los valores límite fijados tanto para las configuraciones como para la cantidad de usuarios. Estos valores fueron establecidos por nosotros, sin posibilidad de que el administrador los modifique ya que tienen que ver con cómo se guardan estos valores y con mantenerlos en un rango razonable.

Cliente del protocolo (Administrador)

Para demostrar el funcionamiento del monitoreo y configuración del proxy, creamos una aplicación cliente que intenta conectarse a la dirección y puerto de monitoreo y configuración del proxy. Esta aplicación recibe todas las configuraciones y comandos por argumentos. Decidimos no hacerla interactiva por cuestiones de simplicidad.

La aplicación recibe como argumento obligatorio un usuario y contraseña para autenticarse al servidor, los cuales deben corresponderse con un usuario y contraseña de nivel administrador registrado en el proxy. En caso de poder establecer la conexión y lograr autenticarse correctamente, procede a enviar los comandos también listados como argumentos.

Por cada comando ejecutado, imprime un mensaje indicando el resultado de la operación y, en caso de corresponder, el valor pedido. Se enviará cada uno de los comandos ingresados, a lo que el servidor irá respondiendo en orden hasta terminar o encontrar un error irrecuperable.

El cliente está hecho de forma tal de validar únicamente que los argumentos recibidos tengan el formato correcto (por ejemplo, si el parámetro debe ser un número). Pero si uno ingresa algún valor incorrecto, como puede ser alguna métrica inexistente, se permite el envío para mostrar cómo responde el server ante estos errores.

Además, se muestra por un lado, los errores que tira el cliente. Y por el otro, los que envía el server. De esta manera, podemos distinguir correctamente entre unos y otros.

| Problemas encontrados

El primer problema que tuvimos fue con la realización de un archivo Makefile que funcione correctamente, generando un ejecutable para el servidor y uno por archivo de testing, y recompilando archivos únicamente si hubo cambios. Luego de varios intentos por hacerlo manualmente, optamos por utilizar la herramienta CMake para generar los archivos Makefile correspondientes. Nunca la habíamos utilizado previamente y terminó siendo extremadamente útil y sencilla de configurar.

El segundo problema surgió al momento de desarrollar el estado de copia de la máquina de estados general. Tuvimos dificultades para determinar los intereses iniciales de cada extremo, así como para cerrar el intercambio de forma prolija, utilizando la syscall shutdown para indicar que un extremo ya no recibirá más información. Aquí también aparecieron algunos códigos de error como ECONNRESET y ENOTCONN que no sabíamos cómo interpretar. Pero logramos solucionarlo luego de varias preguntas en clase.

Otro inconveniente que surgió fue la aparición de una señal de SIGPIPE luego de usar intensivamente el servidor con Google Chrome. Agregamos un flag de MSG_NOSIGNAL en las llamadas a la syscall send para evitar que se generen.

Por último, tuvimos ciertas dificultades para resolver consultas DNS para direcciones IPv4 e IPv6. En un primer momento, intentábamos resolver ambas consultas a la vez, pidiendo tanto los registros A como los AAAA. La manera que encontramos de resolverlo, mejor explicada en *aplicaciones desarrolladas*, fue hacer las consultas por separado.

| Limitaciones de la aplicación

La primer limitación de nuestra aplicación es la cantidad de usuarios concurrentes permitidos. Decidimos limitar la cantidad de clientes concurrentes a 600 clientes y la cantidad de administradores concurrentes a 400. Esto se debe en primer lugar, a la limitación misma de usar pselect, que limita la cantidad de file descriptors en 1024. Además, según las pruebas realizadas, ya con un número cercano a 500 el proxy tiene un funcionamiento extremadamente lento, por lo que consideramos que no tiene tanto sentido permitir muchos más clientes si no podrán usarlo en niveles aceptables. La cantidad de administradores se fijó en 400 simplemente por la mencionada limitación de pselect.

Otra limitación de la aplicación tiene que ver con los comandos soportados de requests del protocolo SOCKS v5. Solamente se aceptan aquellos requests con el comando CONNECT.

Una limitación particular de nuestra aplicación es la cantidad de usuarios registrados, que tiene un máximo de 65535 usuarios. Esto se hizo para simplificar el armado de la respuesta del server en caso de que un administrador desee listar los usuarios. De esta manera, podemos fijar el byte de cantidad de bytes de NUSERS (NULEN) en 2 y simplificar el armado y parseo de la respuesta.

Con respecto al manejo de errores, muchas veces debimos retornar un código de respuesta GENERAL SOCKS SERVER FAILURE dado que el protocolo no contempla códigos de error específicos más que nada para errores de parseo de requests. También podemos mencionar que en cualquier caso, si hay algún error al intentar alocar memoria, se libera esa conexión y se pasa a un estado de error.

Con respecto al disector de contraseñas, como mencionamos anteriormente, su capacidad de detección es limitada ya que decidimos no perjudicar tanto la performance en favor del mismo. Por ello, solo chequea la primera autenticación en un intercambio de datos.

Por otra parte, cada uno de los valores configurables tiene sus valores límite. El timeout de conexión debe estar entre 5 segundos y 4 minutos. Estos valores fueron determinados para que, durante un funcionamiento normal, no se generen demasiados errores de timeout. Además, el límite superior se encuentra ya que, en un valor cercano a ese tiempo, la misma syscall de connect falla como

máximo en caso de no saber cómo llegar hasta un host. Por ello, no tiene sentido que sea superior. Su valor inicial es de 30 segundos.

El timeout general debe estar entre 5 y 60 minutos. El mínimo se fijó en tal valor ya que no es correcto que éste sea mayor al timeout de conexión, ya que no se manejaría correctamente el error de conexión. El máximo es tal que permita estar un tiempo importante sin hacer ninguna operación (como, por ejemplo, puede estarlo YouTube en caso de poner pausa) sin que se permitan conexiones "muertas" indefinidamente. Su valor inicial es de 30 minutos.

Los buffers de los administradores se fijaron en un valor de 4kB sin posibilidad de modificaciones. Este valor permite encadenar varias operaciones de configuración sin consumir recursos de forma excesiva en los mismos. Es por ello que decidimos que no sea configurable.

Los buffers de funcionamiento del proxy socks deben estar entre 2kB y 512kB. El mínimo se seteo intentando hacer un única descarga de manera local. Si el tamaño de buffer era menor, el rendimiento ya era bastante peor al obtenido sin el proxy. Seteamos su valor inicial en 8kB , ya que con este valor obtuvimos los mejores resultados con JMeter (más adelante se adjuntan los gráficos).

| Posibles extensiones

Una posible mejora es utilizar pools de estructuras socks5 y de respuestas de DNS, de forma tal de reutilizar los recursos ya reservados previamente en lugar de destruir las estructuras que ya no se utilizan y construir de cero cada vez que se necesita una nueva.

En lo que refiere al cliente DOH hay una serie de mejoras que se podrían realizar. Primero, se podría utilizar conexiones persistentes y pipelining, lo que permitiría realizar menos conexiones, evitando así el overhead de las mismas, y enviar mas request juntas, reduciendo el tiempo total que tardan. También se podrían cachear los resultados de los requests, reduciendo la cantidad de los mismos que debe hacer el servidor. Por último, se podría expandir la cantidad de test, dado que el mismo no posee ninguno (aunque fue fuertemente testeado).

Con respecto al protocolo, éste fue hecho de manera tal que sea fácilmente extensible. Es por ello que se podrían agregar nuevas métricas y configuraciones sin problemas en una segunda versión del mismo. Por ejemplo, se podría tomar una métrica de tiempo promedio de conexión o medir la velocidad promedio del proxy.

Usando una métrica como la mencionada, se podría ajustar dinámicamente los tiempos de timeout, de forma tal de que el timeout configurado sea un "ratio" y que, según conexiones previas, se configure uno u otro valor de timeout para las conexiones.

| Conclusiones

Finalmente, se logró un funcionamiento correcto del proxy y del cliente realizado acorde a lo que se esperaba de los mismos. Fue muy importante lograr hacer funcionar cada módulo por separado, testeando cada parser con su test unitario en la medida de lo posible y armando la máquina de estados general del parser básica, sin todos los demás requisitos. Sobre esto, se fue trabajando, unificando los diversos agregando la demás funcionalidad pedida, para alcanzar la versión final.

| Ejemplos de prueba

Se expuso al proxy a la descarga de diversos archivos de diferentes tamaños, siendo a su vez sometidos a diferentes números de conexiones concurrentes. En estas condiciones, en base al tiempo de cada una, obtuvimos el tiempo promedio en que demoró en concretarse en cada caso. Experimentamos bajo la resolución DNS del curl, y suprimiendo esa herramienta. Al mismo tiempo, las mismas pruebas se hicieron para un archivo mucho mayor creado de forma local mediante:

```
dd if=/dev/urandom of=archivo-in bs=4096 count=64000
```

Los resultados fueron los siguientes:

	Con resolución de curl			**Sin resolución de curl**			**Local**
# concurrentes x archivo	**S (750KB)**	**M (2MB)**	**L (8.5MB)**	**S (750KB)**	**M (2MB)**	**L (8.5MB)**	**archivo-in (260MB)**
3	2,50364"	2,778"	6,98"	2,31"	3,64"	6,5"	6,3"
10	2,3478"	4,232"	13,78"	2,5"	4,4"	12,8"	1,58"
50	5,2118"	11,2622"	57,87"	5,48"	11,6"	57,8"	12,2"
75	6,41"	16,015"	45,3"	6,7"	16,1"	81,3"	30,9"
100	8,07"	21,669"	94,647"	8,53"	21,58"	82,16"	54,2"
500	44,5"	63,1"	40"	50,8"	134,8"	68"	104,8"

Otra prueba realizada fue conectándose con ncat y stty a una dirección local, de forma tal de verificar que funcione correctamente con recepción en el peor caso (recibiendo de a un byte). Para ello corrimos un ncat escuchando en ::1 el el puerto 9090 y nos conectamos desde otra terminal con

```
$ stty -icanon && ncat -C --proxy-type socks5 --proxy-auth peter:123
--proxy 127.0.0.1:1080 ::1 9090
```

Por último, realizamos algunos testeos utilizando JMeter. Para ello, lo corrimos indicando cómo conectarse al proxy (habiendo deshabilitado la etapa de negociación, ya que no pudimos configurarla con JMeter)

./jmeter.sh -DsocksProxyHost=localhost -DsocksProxyPort=1080

Lo probamos con 4 casos, según el tamaño de ambos buffer: 8kB, 16kB, 32kB y 64kB. A continuación, adjuntamos las métricas obtenidas al momento de realizar las pruebas y los gráficos de los datos obtenidos en cada caso.

8kB

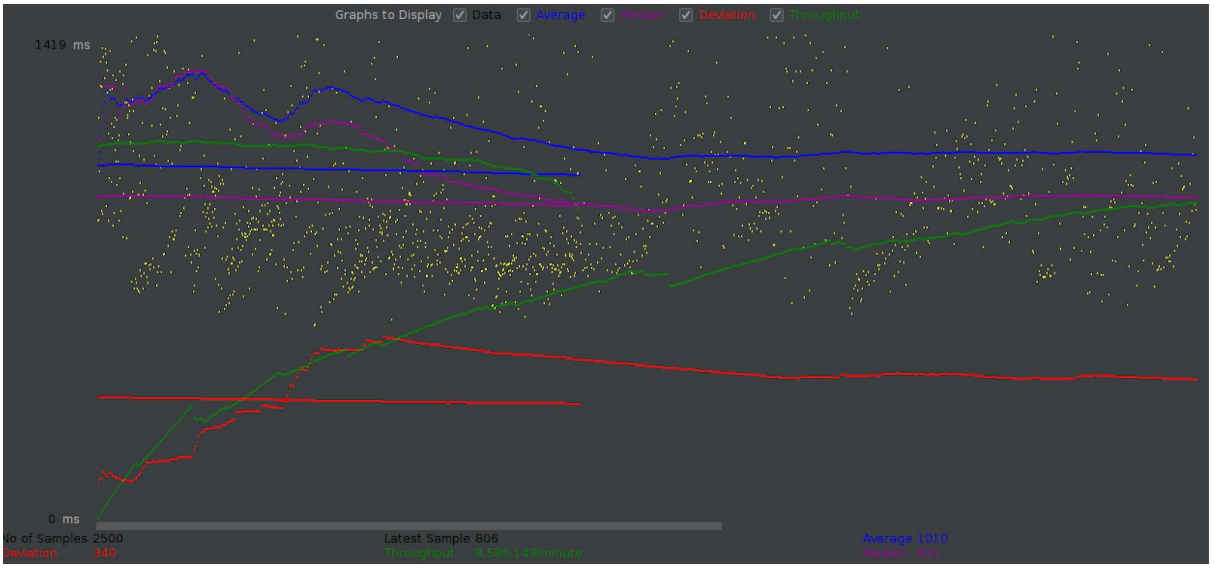
```
→  pc-2020a-1 git:(develop) ✗ ./build/src/client -u admin:admin +get-config 0 +get-config 1 +get-config
 2 +get-config 3 +get-metric 0 +get-metric 1 +get-metric 2

---------------- Start Server Responses  ----------------

Server -> Read buffer size: 8000 bytes
Server -> Write buffer size: 8000 bytes
Server -> General timeout: 1800 s
Server -> Connection timeout: 30 s
Server -> Historical connections: 346
Server -> Concurrent connections: 115
Server -> Historical byte transfer: 18257966 bytes

---------------- End Server Responses    ----------------

→  pc-2020a-1 git:(develop) ✗
```



16kB

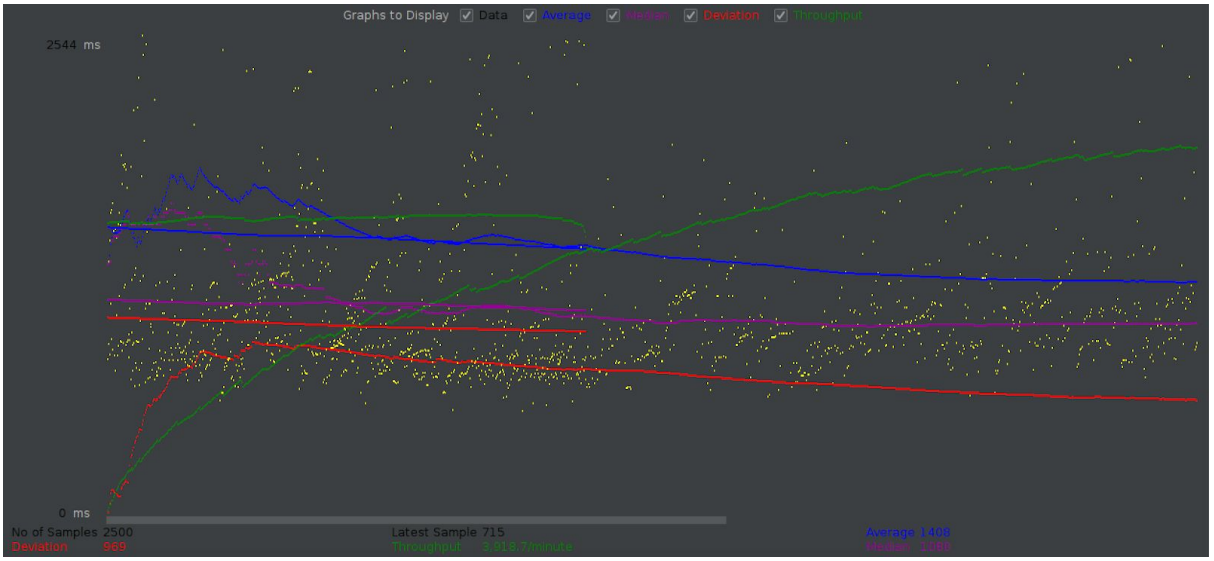
```
→  pc-2020a-1 git:(develop) ✗ ./build/src/client -u admin:admin +get-config 0 +get-config 1 +get-config
 2 +get-config 3 +get-metric 0 +get-metric 1 +get-metric 2

---------------- Start Server Responses  ----------------

Server -> Read buffer size: 16000 bytes
Server -> Write buffer size: 16000 bytes
Server -> General timeout: 1800 s
Server -> Connection timeout: 30 s
Server -> Historical connections: 1494
Server -> Concurrent connections: 263
Server -> Historical byte transfer: 61010643 bytes

---------------- End Server Responses    ----------------

→  pc-2020a-1 git:(develop) ✗
```



32kB

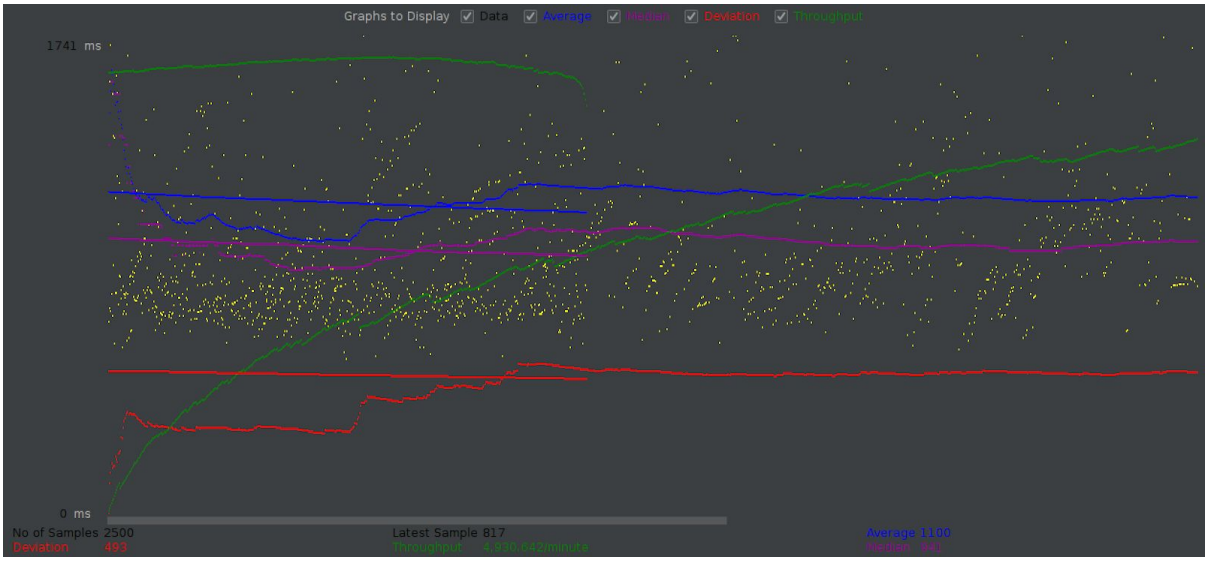
```
➜  pc-2020a-1 git:(develop) ✗ ./build/src/client -u admin:admin +get-config 0 +get-config 1 +get-config
 2 +get-config 3 +get-metric 0 +get-metric 1 +get-metric 2

---------------- Start Server Responses  ----------------

Server -> Read buffer size: 32000 bytes
Server -> Write buffer size: 32000 bytes
Server -> General timeout: 1800 s
Server -> Connection timeout: 30 s
Server -> Historical connections: 2419
Server -> Concurrent connections: 188
Server -> Historical byte transfer: 81028752 bytes

---------------- End Server Responses    ----------------

➜  pc-2020a-1 git:(develop) ✗
```



64kB

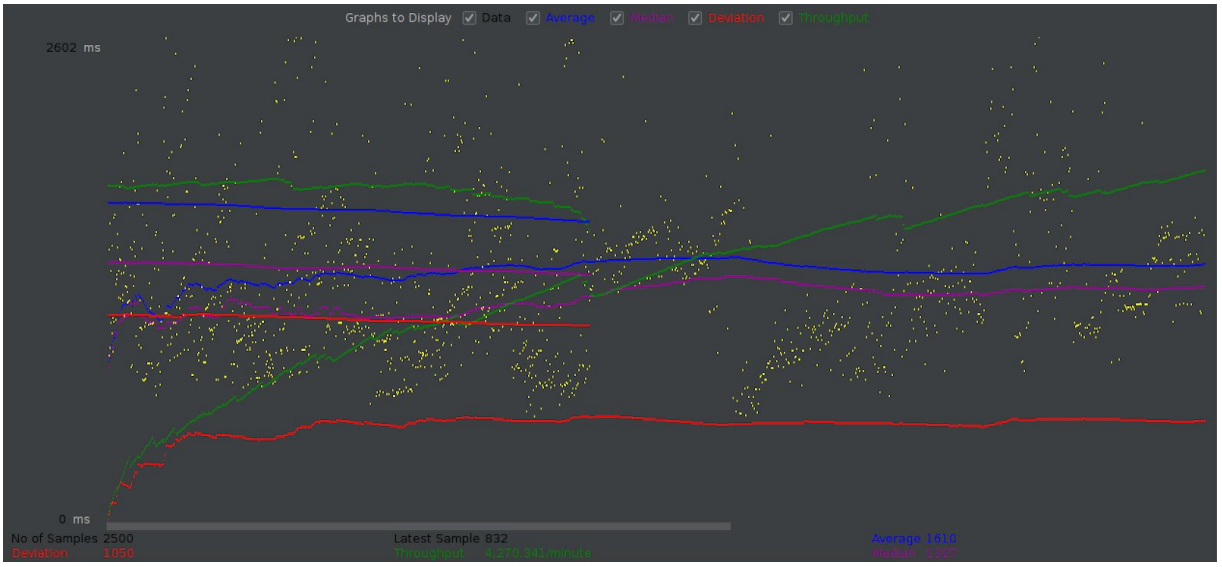
```
→  pc-2020a-1 git:(develop) ✗ ./build/src/client -u admin:admin +get-config 0 +get-config 1 +get-config
 2 +get-config 3 +get-metric 0 +get-metric 1 +get-metric 2

---------------- Start Server Responses  ----------------

Server -> Read buffer size: 64000 bytes
Server -> Write buffer size: 64000 bytes
Server -> General timeout: 1800 s
Server -> Connection timeout: 30 s
Server -> Historical connections: 2967
Server -> Concurrent connections: 236
Server -> Historical byte transfer: 91301951 bytes

---------------- End Server Responses    ----------------

→  pc-2020a-1 git:(develop) ✗
```



| Guía de instalación

A continuación, se describen los pasos necesarios para generar los archivos ejecutables de ambas aplicaciones desarrolladas a partir del archivo tarball. Deberá tener instalada la herramienta “cmake”, al menos en su versión 2.8.

1. Descargar el archivo tarball pc-2020a-1.tar.gz
2. Descomprimir el tarball utilizando el comando tar
   tar xzf pc-2020a-1.tar.gz
3. Ingresar al directorio “pc-2020a-1”
4. Crear el directorio “build”
5. Ingresar al directorio “build” y abrir una terminal aquí
6. Generar los archivos de configuración con cmake utilizando
   cmake ..
7. Vuelva al directorio raíz
   cd ..
8. Genere los archivos ejecutables con make utilizando

make -C build clean all

Ambas aplicaciones ya están listas para ser ejecutadas.

Para correr el proxy socks, abra una terminal en el directorio “pc-2020a-1” y ejecute

./build/src/socks5d

Puede encontrar las distintas opciones de ejecución en el manual socks5d, el cual puede abrir ejecutando

man ./socks5d.8

Una vez que el servidor está corriendo, ya puede utilizar la aplicación que desee que soporte un proxy SOCKS v5 con autenticación. Si quisiéramos ya tener usuarios cargados al ejecutarlo, podemos agregarlos con -u o -U como indica el manual o bien crear un archivo “users.txt” en el directorio “pc-2020a-1” con las credenciales en el formato *username:password:level*, siendo level el nivel de acceso del usuario. Como se explica en el manual, los niveles de acceso son cliente (nivel 0) y administrador (nivel 1).

Para correr el cliente administrador, abra una terminal en el directorio “pc-2020a-1” y ejecute

./build/src/client

Puede encontrar las distintas opciones de ejecución en el manual client, el cual puede abrir ejecutando

man ./client.8

| Instrucciones para la configuración

Por un lado, el server recibe configuraciones al iniciarse por argumentos. De esta manera, se configuran los datos del servidor doh, las direcciones y puertos donde servirán el proxy y el servicio de management, y si el disector de contraseñas está habilitado o no. Sus instrucciones de uso se encuentran en el manual del proxy, al cual se puede acceder ejecutando abriendo el archivo socks5d.8 con man.

Además, el proxy puede ser monitoreado y configurado desde una aplicación cliente de un administrador, como la desarrollada por nosotros. Sus instrucciones de uso se encuentran en el manual de la aplicación, al cual se puede acceder ejecutando abriendo el archivo client.8 con man.

| Ejemplo de configuración y monitoreo

Un ejemplo de uso del cliente puede ser el siguiente:

| Documento de diseño del proyecto (Arquitectura de la aplicación)