Protocolos de Comunicación

20/06/2022

Alberto Abancens 62581

Gonzalo Rossin 60135

Uriel Mihura 59039

Índice

Descripción de los protocolos y aplicaciones desarrolladas

Se desarrolló primero un servidor que funciona como Proxy SOCKS5. Esto significa que un cliente puede conectarse a un servidor remoto, utilizando nuestro servidor como intermediario. Así, el servidor remoto verá que el que se conecta es nuestro servidor y no el cliente. Así también, la conexión saliente del cliente será a nuestro servidor y no al servidor remoto.

Al mismo, se le agregaron funcionalidades adicionales; recolección sobre estadísticas de uso, configuración de distintas variables, y la implementación de un "disector" de credenciales POP3. Esto significa que cualquier cliente que decida conectarse a un servidor POP3 mediante nuestro proxy, dejará en el mismo una copia de sus credenciales.

Luego, se desarrolló un cliente "Administrador" (o "Admin") que se conecta a nuestro servidor por otro puerto y con otro método de autenticación. Este Admin sirve, no para usar el servidor como un proxy, sino para poder editar y monitorear el mismo. Para esta comunicación, se desarrolló nuestro propio Protocolo de Comunicación, llamado SSEMD: Socks Server Editor Monitor & Disector, el cual está descrito en el RFC adjunto en este proyecto.

El protocolo SSEMD se basa en un Request-Response entre el Admin y el Servidor. El Admin le manda un mensaje indicando la versión, el token de autenticación, si quiere "Get" o "Edit" algo del servidor, qué es lo que quiere obtener o editar, y opcionalmente alguna información extra (por ejemplo al querer agregar un nuevo usuario, se deberá proveer su [usuario:contraseña]).

Problemas encontrados durante el diseño y la implementación

El servidor pasó por varias etapas. En un principio se comenzó por un servidor TCP cuya función era responder un echo request. La razón de esto era no iniciar con un "pizarrón en blanco" y construir en base a ello lo que posteriormente sería nuestro servidor.

El primer objetivo fue que el cliente se conecte, de manera bloqueante, al Servidor, y que éste se conecte a una página fija de internet, para que el Servidor le mande automáticamente la respuesta al cliente, haciendo así un proxy transparente.

El primer problema grande fue hacer que el Servidor permita más de una conexión a la vez, y que estas puedan ser por IPv4 o IPv6. No podíamos establecer un socket pasivo en IPv6 al mismo tiempo que en IPv4, y descubrimos que se debía a que por defecto el bind de un socket IPv6 intenta utilizar también la región IPv4. Para solucionarlo hubo que setear una opción luego de crear el socket.

Una vez desarrollada la versión proxy transparente bloqueante del trabajo intentamos incorporar el select para hacerlo no bloqueante. Esto nos trajo muchos problemas pero nos ayudó a entender mucho mejor el esqueleto del proyecto. El siguiente paso fue reemplazar lo que teníamos hecho por la abstracción del selector provista por la cátedra, que si bien tomó un par de días de adaptar fue la decisión correcta porque hizo al proyecto mucho más robusto y legible. En este punto logramos tener un proxy transparente que únicamente se bloqueaba en la resolución de nombres pero decidimos avanzar con la implementación de SOCKS5 antes.

Para ello partimos del análisis del RFC realizado en clase para definir los estados de una conexión. Tomamos la decisión de no utilizar la máquina de estados provista por la cátedra y en su defecto utilizar switch en los handlers y un manejo de estados menos sofisticado. Si bien no terminó siendo la solución más elegante, nos permitió aprender y entender mucho más el paso a paso del desarrollo.

En este punto uno de los integrantes ya había estado desarrollando los primeros parsers. En nuestro caso arrancamos con hello y request dejando el de autenticación para el final. El estado de Hello una vez ajustado el parser fue sencillo de integrar pero no lo fue tanto el de request. El flujo de la conexión se complica un poco en esta parte. Primero hicimos el caso completo para IPv4. Nos topamos con problemas de manejo de las estructuras de la familia sockaddr, casteos y demás, pero siguiendo el ejemplo visto en clase, salió. Luego pasamos al caso de resolución de nombres.

En este punto cabe destacar que a lo largo del desarrollo del proyecto el mal manejo de la memoria dinámica nos costó mucho tiempo, generando errores bastante crípticos y difíciles de debuggear. Nos pasaron todo tipo de cosas, pisamos punteros a función, estructuras y muchos segmentation faults.

Fue difícil probar que la resolución de nombres preguntara por otras ips en caso de fallar la primera. Una forma que se nos ocurrió es levantar un nginx y en el etc/hosts decir que bar

resuelva a localhost tanto IPv4 como IPv6. Interferimos con el debugger haciéndolo fallar el connect a propósito y verificando que pruebe con la siguiente ip.

En este punto del desarrollo nos dimos cuenta que el estado de copia estaba funcionando muy lento y resulta ser que los intereses de escritura y lectura se estaban asignando sin mucho criterio. Terminamos por hacer una implementación mucho más prolija de todo él estado, con funciones como *determine_interest()* que le dan bastante claridad al código.

A esta altura teníamos un SOCKS5 funcionante que procesaba curls y podía proveer internet tanto a firefox como a google-chrome.

Lo siguiente fue agregar la autenticación de usuario situada entre los estados *hello* y *request*, que con la experiencia de los demás estados y parsers no produjo mucho problema. Con curl andaba de lo mas bien pero con chrome y firefox no. Después de investigar un poco descubrimos que ninguno de ellos implementa la versión 5 del protocolo Socks con autenticación.

Continuamos por crear el parser de POP3 e involucrarlo en el estado de copia para hacer la disección de contraseñas. Para que no haya conflictos con los otros buffers se usó uno auxiliar para realizar este trabajo. Para probarlo utilizamos nc y levantamos el dovecot en local. Ver a esto funcionar fue una gran satisfacción, ya estaba tomando buena forma el proyecto.

El siguiente objetivo fue crear el cliente Admin, que se pueda conectar al servidor por otro puerto, y que el servidor maneje sus pedidos y le responda lo apropiado. Todo siguiendo los lineamientos de nuestro RFC propuesto.

Al principio el Admin mandaba un *request* igual a un *get* http, entonces era tratado como otro *Client*, pero se podía conectar. Luego, mandaba un mensaje fijo del estilo de nuestro RFC. En este momento se tenía que darle las herramientas al servidor para que pueda comunicarse usando este nuevo Protocolo de Comunicación, que no fue demasiado complicado ya que era un parser similar al ya implementado para SOCKS5.

Luego se implementó una estructura de parámetros, inspirado por los archivos args.c y args.h provistos por la cátedra, para que el mensaje que envía el Admin no sea fijo, sino dinámico.

Al momento de querer procesar la respuesta para el Admin, se tuvo que modificar un poco la estructura del Servidor, ya que ahora muchos de los parámetros que antes se consideraban fijos, ahora eran variables y configurables por el Admin (*buffsize*, etc.).

La implementación del intercambio de mensajes entre el Server y el Admin resultó problemático. Esto es debido a que se debían tratar datos del tipo uint8_t, y no algo como un int o string que son más legibles por humanos, tornando bastante más dificultoso el proceso de debugging de problemas con por ejemplo *little* y *big endian*.

Más de una vez en este proceso, el Servidor debió sufrir *refactors* grandes a su código, ya que nos quedaban archivos demasiado grandes y poco modularizados debido a que en un principio no se quería lidiar con los problemas generados por el makefile y simplemente garantizar que el código funcionase.

Otro problema encontrado cerca del final del proyecto fue que siempre se tomaban como ejemplo y uso, casos ideales. Lo que llevó más de un día de implementar, responder y avisar los casos de error. (Estaban bien señalados, solo que al toparse con un problema el servidor, cliente y/o admin quedaban en un estado inutilizable). Esto incluye memory leaks, warnings menores ignorados por demasiado tiempo y algún que otro problema menor dejado “para más adelante”.

Otro problema encontrado recién al momento de escribir este informe, fue la falta de ir llenando el informe mientras se avanzaba con el proyecto. Hubiera sido de gran ayuda ir agregando por lo menos 1-2 frases de los problemas encontrados cada día.

Limitaciones de la aplicación

- Una primera limitación que tiene es que no puede manejar a más de 10 *Admins* al mismo tiempo. De todas formas, esto fue una decisión tomada conscientemente, ya que al ser una comunicación del tipo Request-Response, es demasiado rápida como para que en un caso de uso real, 2 administradores se pisen, menos aún 10.

- El servidor permite hasta 10 usuarios, otra decisión tomada conscientemente, porque con 10 es suficiente para ver y probar toda la potencia del programa, y es muy simple cambiar el valor de una variable para poder permitir más usuarios.

- El servidor permite hasta 500 conexiones SOCKS5. Esto es porque se necesitan 2 File Descriptors por cada conexión, y select puede manejar como máximo 1024 File Descriptors, y se necesitan algunos para los sockets pasivos y para el manejo del Admin. Así, alcanza para más de 500 clientes, pero se decidió que dejarlo en 500 no era mala idea.

- Por más que se pueda editar el Timeout del selector, no ocurre nada especial cuando esto sucede. Podría implementarse algún sistema del tipo “garbage collector” para ir desconectando a los clientes que llevan demasiado tiempo sin actividad.

Posibles extensiones

1. Extender el disector del servidor, para que registre credenciales de http o de cualquier otro protocolo que no esté encriptado.

2. Guardar dichas credenciales en un archivo para que no se pierdan, o en alguna estructura de datos para que sean accesibles por el administrador.

3. Guardar usuarios y contraseñas, estadísticas y configuraciones para que no se pierdan al reiniciar el servidor.

4. Extender el RFC para que soporte más funcionalidades y estadísticas, por ejemplo específicas de cada usuario.

5. Una “lista negra” de usuarios para que no puedan utilizar el servidor, ya sea por IP o por [user:password] y que no puedan ser agregados nuevamente.

6. El cliente del administrador podría ser no bloqueante. Ahora es bloqueante porque éste solo debe interpretar sus argumentos para formar el mensaje apropiado, mandarlo, y leer su respuesta.

7. El administrador podría ser un usuario registrado en el servidor, más que un token.

8. Un sistema tipo garbage collector, para que los usuarios con inactividad sean desconectados.

Diseño del proyecto

Se decidió utilizar TCP como protocolo de transporte ya que nos resultó más sencillo de programar dado que solo hay que tratar un flujo de bytes y no hay que preocuparse por calcular los datos que podemos mandar en un datagrama y en caso de no entrar, separar el mensaje en varios datagramas. Por otra parte, el uso de TCP nos pareció más adecuado debido a la naturaleza de las operaciones que nuestro servidor debía llevar a cabo como manejar la comunicación entre cliente y servidor por medio de un browser como google chrome a través de nuestro proxy y la transferencia de archivos.

Se decidió un buffer inicial de 4096 bytes, ya que con 500 clientes teniendo 2 buffers cada uno, termina siendo 4MB de memoria, lo cual es aceptable.

Se decidió un timeout del selector inicial de 10 segundos, ya que da tiempo a que un humano pueda mandar algo si lo quiere manualmente, pero no es tanto como para que parezca que hay algún bloqueo en alguna parte.

El proyecto comienza su ejecución en el archivo Server.c . Al principio se parsean los argumentos pasados por línea de comando, y se crean las primeras estructuras acordemente (clientes, etc). Decidimos que es obligatorio pasar un token (con el argumento -t) para que el Admin se pueda autenticar al conectarse.

Luego, se crean los Master Sockets, pasivos, IPv4 e IPv6, que servirán para atender a las conexiones entrantes, tanto para SOCKS como para SSEMD.

Luego, se crea el selector(), que estará encargado de manejar los pedidos de forma multiplexada no bloqueante. Al mismo se le registran los File Descriptors de los Sockets pasivos, con sus handlers correspondientes.

Aquí es donde se entra en el ciclo infinito para continuamente llamar al selector, en un principio esperando solamente a conexiones entrantes.

Al momento de recibir una conexión, el selector llama al MasterSocks5Handler que se ocupará de inicializar el nuevo cliente (datos relevantes, buffers, etc) y registrarlo para lectura en el selector.

Para cada estado de la conexión, los handlers de lectura y escritura se encargarán de ejecutar la función correspondiente (leyendo o escribiendo del File descriptor apropiado). Los distintos estados son, en el siguiente orden: hello_read, hello_write, up_read, up_write, request_read, request_resolv, request_connecting, request_write y connected_state. Una vez conectado, el proxy recibe los bytes del cliente y los envía (pasando por el POP3 disector) al servidor remoto, como también recibe los bytes del servidor remoto y se los envía al Cliente. Cada una de estas funciones se ejecuta en iteraciones distintas del selector(), únicamente si el File descriptor tenía asignados los intereses de lectura y/o escritura adecuados.

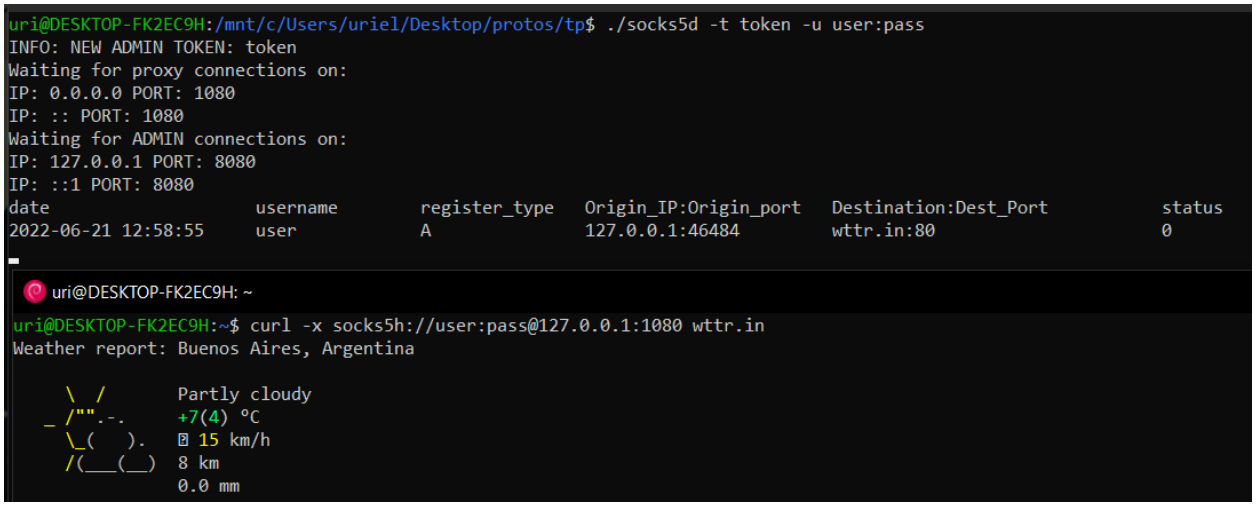
Al terminarse una conexión, se libera la memoria ocupada y se desocupa el cliente para que lo pueda utilizar una nueva conexión entrante.

El manejo del Admin es distinto. Éste primero arma el mensaje a ser enviado parseando los argumentos recibidos, y luego envía el mensaje por TCP al puerto apropiado. El Servidor, al recibir un mensaje así, lo parsea nuevamente, verifica que el token y los argumentos sean correctos, y ejecutará una función apropiada para procesar el pedido. Luego, arma un mensaje de respuesta y se lo envía de vuelta al Admin. El Admin interpreta la respuesta e imprime en pantalla su significado.

Ejemplos de prueba

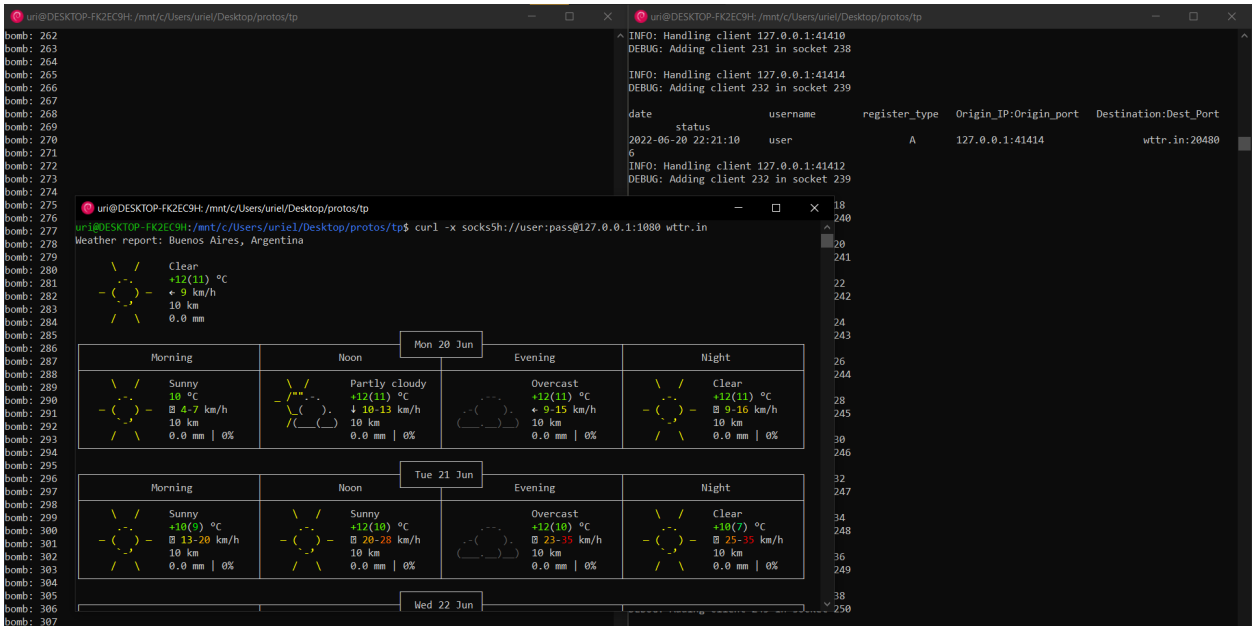
Ejecución de un curl usando nuestro proxy:

En la terminal de arriba se ejecuta el Servidor y en la de abajo se ejecuta un curl usando nuestro proxy como intermediario. Se puede ver el registro de entrada en el Servidor.



Para probar que se podían manejar cientos de usuarios al mismo tiempo, se implementó un pequeño archivo que inicia conexiones con el Servidor. Usando éste, se pudo probar que teniendo cientas de sesiones iniciadas, el servidor seguía funcionando, aunque notablemente más lento ya que debía iterar sobre todas las conexiones existentes:

En la siguiente captura de pantalla se puede observar cómo en la terminal izquierda un programa inicia cientos de conexiones; en la derecha el Servidor manejando las nuevas conexiones, como manejando el cliente de la terminal del medio, un CURL a wttr.in :



Para conseguir el tamaño de buffer apropiado para el default del servidor se realizaron pruebas utilizando curl y nginx. En bar pusimos un archivo de 1MB de tamaño y probamos distintos valores de buffer hasta que la diferencia con curl sea imperceptible.

Curl sin proxy

```
tataabancens@DESKTOP-374DA52: ~
tataabancens@DESKTOP-374DA52:~$ time curl bar | md5sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1024M  100 1024M    0     0   414M      0  0:00:02  0:00:02 --:--:--  414M
cd573cfaace07e7949bc0c46028904ff  -

real    0m2.478s
user    0m2.139s
sys     0m1.893s
```

Ahora a traves del proxy

Buffer: 16KB

```
tataabancens@DESKTOP-374DA52:~$ time curl -x socks5://user:user@127.0.0.1:1080 bar | md5sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1024M  100 1024M    0     0   381M      0  0:00:02  0:00:02 --:--:--  381M
cd573cfaace07e7949bc0c46028904ff  -

real    0m2.690s
user    0m2.407s
sys     0m1.977s
```

Buffer: 4KB

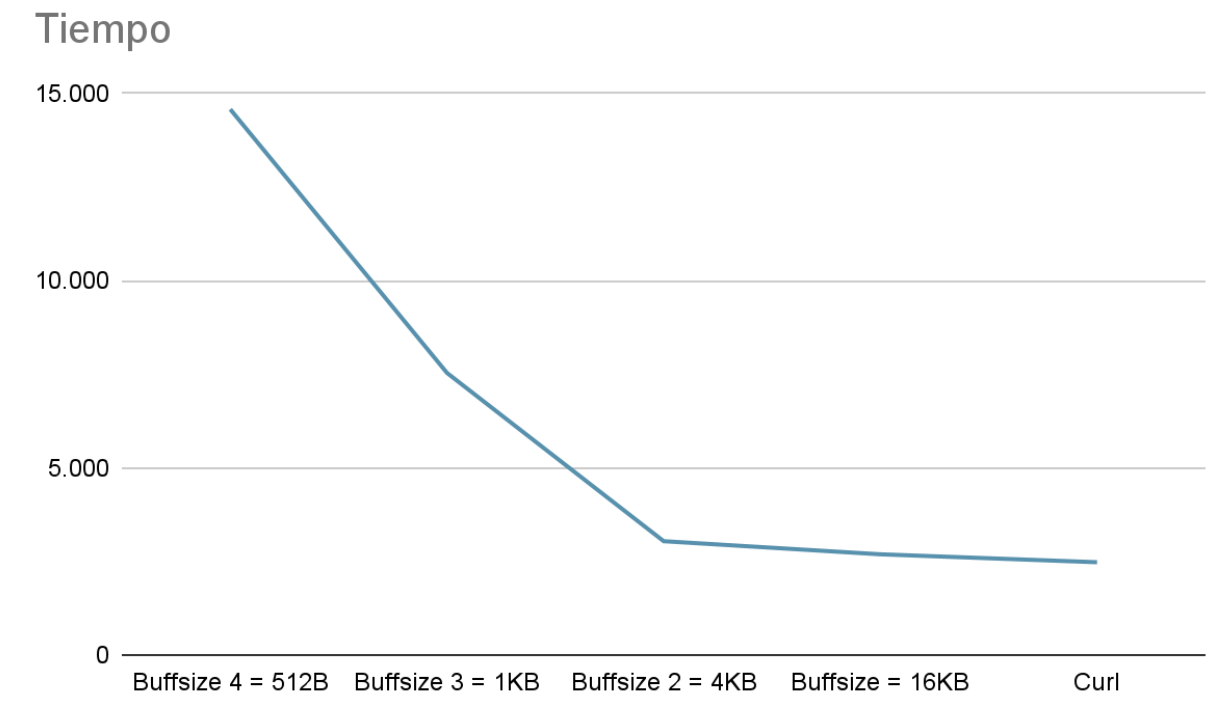
```
tataabancens@DESKTOP-374DA52:~$ time curl -x socks5://user:user@127.0.0.1:1080 bar | md5sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1024M  100 1024M    0     0   337M      0  0:00:03  0:00:03 --:--:--  337M
cd573cfaace07e7949bc0c46028904ff  -

real    0m3.040s
user    0m2.637s
sys     0m2.268s
```

Buffer: 1024B

```
tataabancens@DESKTOP-374DA52:~$ time curl -x socks5://user:user@127.0.0.1:1080 bar | md5sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1024M  100 1024M    0     0   136M      0  0:00:07  0:00:07 --:--:--  138M
cd573cfaace07e7949bc0c46028904ff  -

real    0m7.536s
user    0m2.991s
sys     0m3.298s
```

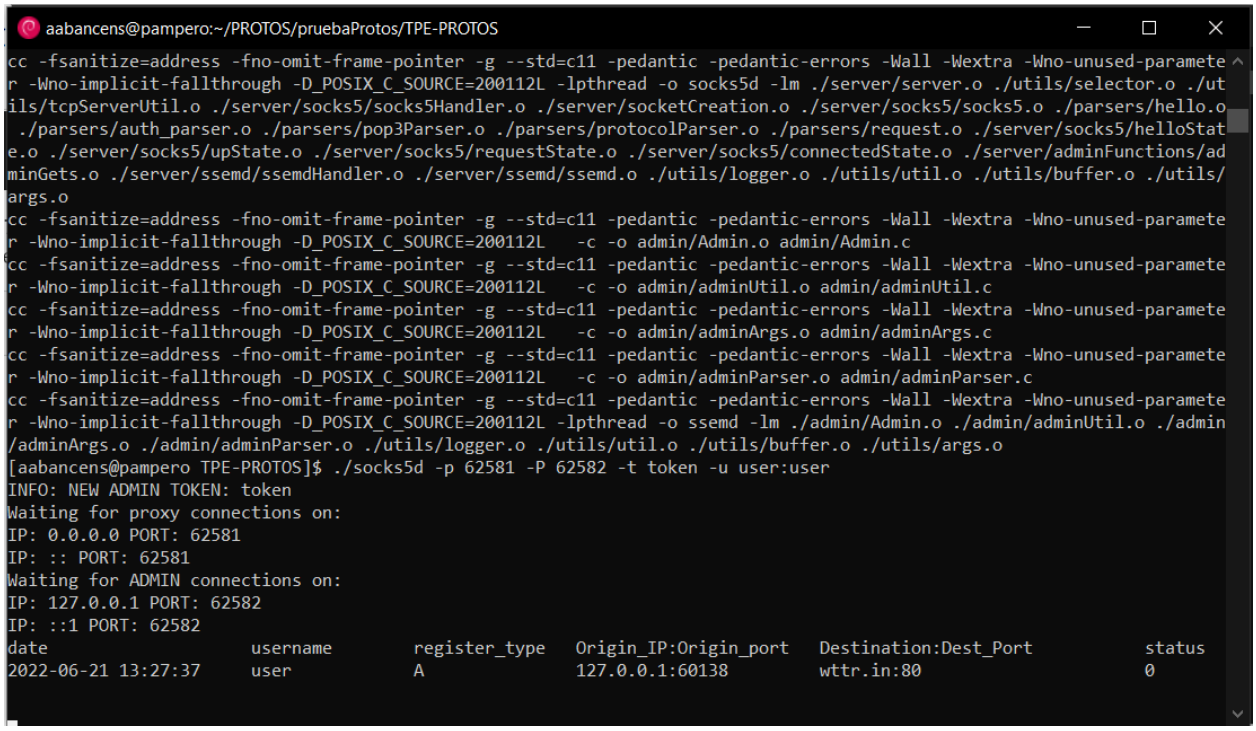
Buffer: 512B

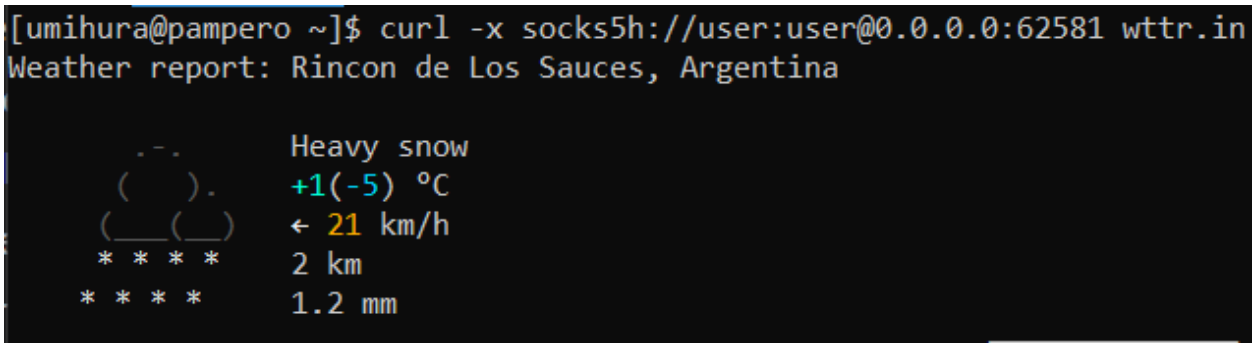
```
tataabancens@DESKTOP-374DA52:~$ time curl -x socks5://user:user@127.0.0.1:1080 bar | md5sum
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1024M  100 1024M    0     0  70.2M      0  0:00:14  0:00:14 --:--:-- 70.9M
cd573cfaace07e7949bc0c46028904ff  -

real    0m14.578s
user    0m4.891s
sys     0m5.014s
```



Luego de revisar la evidencia terminamos por elegir un tamaño default de buffer de 16 KB, teniendo en cuenta que la cantidad máxima de conexiones es 500 que implicaría 1000 buffers., resultando en un uso de 16MB de memoria que para las computadoras de hoy no es una cantidad costosa.

Otro ejemplo de uso es compilar y correr el servidor desde Pampero. Se puede ver en la primer captura como un integrante compila y ejecuta el Servidor mientras que en la segunda captura desde otro usuario se puede usar el proxy normalmente.





Guia de instalación

Una vez teniendo los archivos entregados, deberá ejecutar un make all para generar los ejecutables socks5d y ssemd en la raíz del proyecto:

```
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ls -la
total 148
drwxrwxrwx 1 uri uri   512 Jun 21 10:50 .
drwxrwxrwx 1 uri uri   512 Jun  1 00:21 ..
drwxrwxrwx 1 uri uri   512 Jun 21 10:50 admin
-rwxrwxrwx 1 uri uri 88336 Jun 21 10:31 bomb
drwxrwxrwx 1 uri uri   512 Jun 21 10:49 .git
-rwxrwxrwx 1 uri uri  6596 Jun 20 23:04 .gitignore
drwxrwxrwx 1 uri uri   512 Jun 21 10:47 include
-rwxrwxrwx 1 uri uri 45573 Jun 20 23:00 Informe.pdf
-rwxrwxrwx 1 uri uri  2219 Jun 21 10:47 Makefile
drwxrwxrwx 1 uri uri   512 Jun 21 10:47 notes
drwxrwxrwx 1 uri uri   512 Jun 21 10:50 parsers
drwxrwxrwx 1 uri uri   512 Jun 19 19:26 patches
-rwxrwxrwx 1 uri uri    51 Jun 20 23:00 README.md
drwxrwxrwx 1 uri uri   512 Jun 21 10:50 server
drwxrwxrwx 1 uri uri   512 Jun 19 19:26 tests
drwxrwxrwx 1 uri uri   512 Jun 21 10:50 utils
drwxrwxrwx 1 uri uri   512 Jun 20 12:40 .vscode
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ make all
```

Se producirá texto correspondiente al Makefile haciendo su trabajo, pero ningún warning.

Al terminar, el Make habrá generado 2 archivos en la raíz:

```
-rwxrwxrwx 1 uri uri 345648 Jun 21 10:50 socks5d
-rwxrwxrwx 1 uri uri 135696 Jun 21 10:50 ssemd
```

El archivo socks5d es el Servidor, y el ssemd es el Admin. Ambos cuentan con la opción -h y -v para ver cómo ejecutar los mismos.

Como un extra, se implementó un pequeño programa, bomb, que genera muchas conexiones a la dirección y puerto asignadas en el programa. Para generarlo se deberá ejecutar make bomb, creando el archivo bomb en la raíz. Se ejecuta sin parámetros.

Instrucciones para la configuración

Una vez generados los archivos, tanto el Servidor como el Admin pueden (y deben) tomar argumentos para cambiar su configuración interna, estos pueden ser vistos con el argumento -h.

El Servidor puede editar desde línea de comando: su lista de usuarios inicial, el estado inicial del POP3 dissector, los puertos y direcciónes IPv4 e IPv6 que usará para atender conexiones entrantes y el token que utilizará el Admin para autenticarse.

El Admin puede editar desde línea de comando: su token, si quiere -G get una métrica o -E edit una configuración, seguido de el número de función que se desea ejecutar, y la dirección y puerto en donde se desea enviar el mensaje.

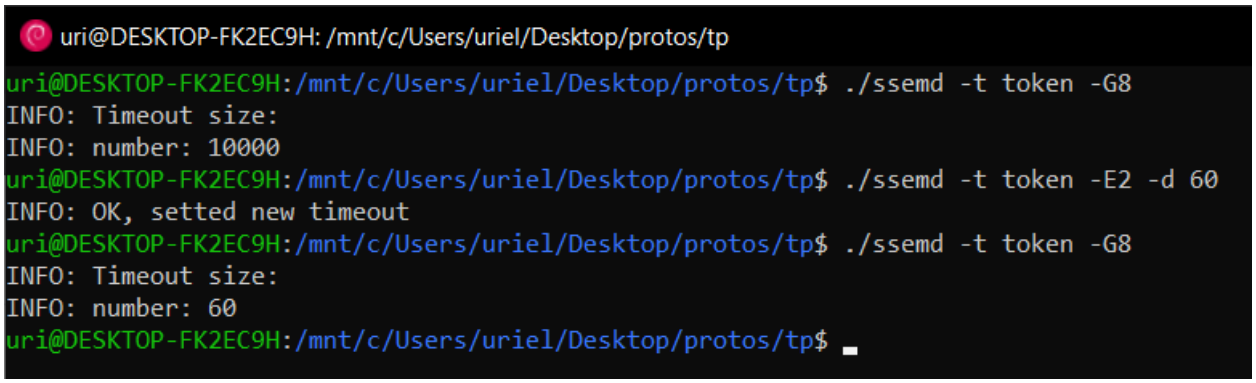
Una vez ejecutado el Servidor, el Admin puede continuamente configurarle distintas variables, la lista de posibles configuraciones es:

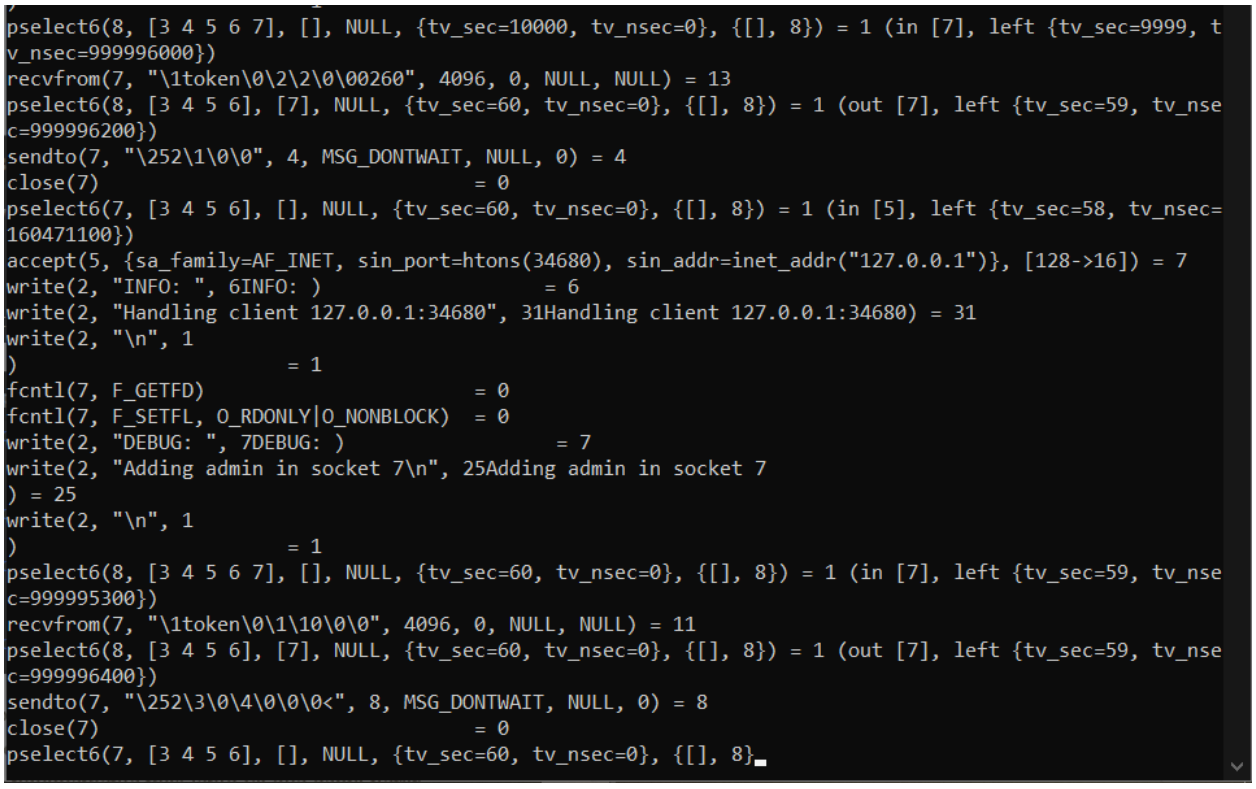
./ssemd -t token -E1 -d 1024	se editará la el buffer size a 1024 bytes.
./ssemd -t token -E2 -d 100	se editará la el timeout del select a 100 segundos.
./ssemd -t token -E3	se encenderá el disector de contraseñas.
./ssemd -t token -E4	se apagará el disector de contraseñas.
./ssemd -t token -E5 -d user:pass	se agregará user:pass a la lista de usuarios.
./ssemd -t token -E6 -d user:pass	se eliminará user:pass de la lista de usuarios.
./ssemd -t token -E7	se encenderá la autenticación con contraseña.
./ssemd -t token -E8	se apagará la autenticación con contraseña.

Ejemplos de configuración y monitoreo

Edición del timeout:

En la primer captura de pantalla se puede ver cómo se obtuvo el valor del timeout, como se edita y como se vuelve a obtener con el valor actualizado. En la segunda captura de pantalla se puede observar cómo el servidor (ejecutado con STRACE) en un principio tiene el pselect con un timeout de 10000 y luego se actualiza a 60.





Edición de usuarios:

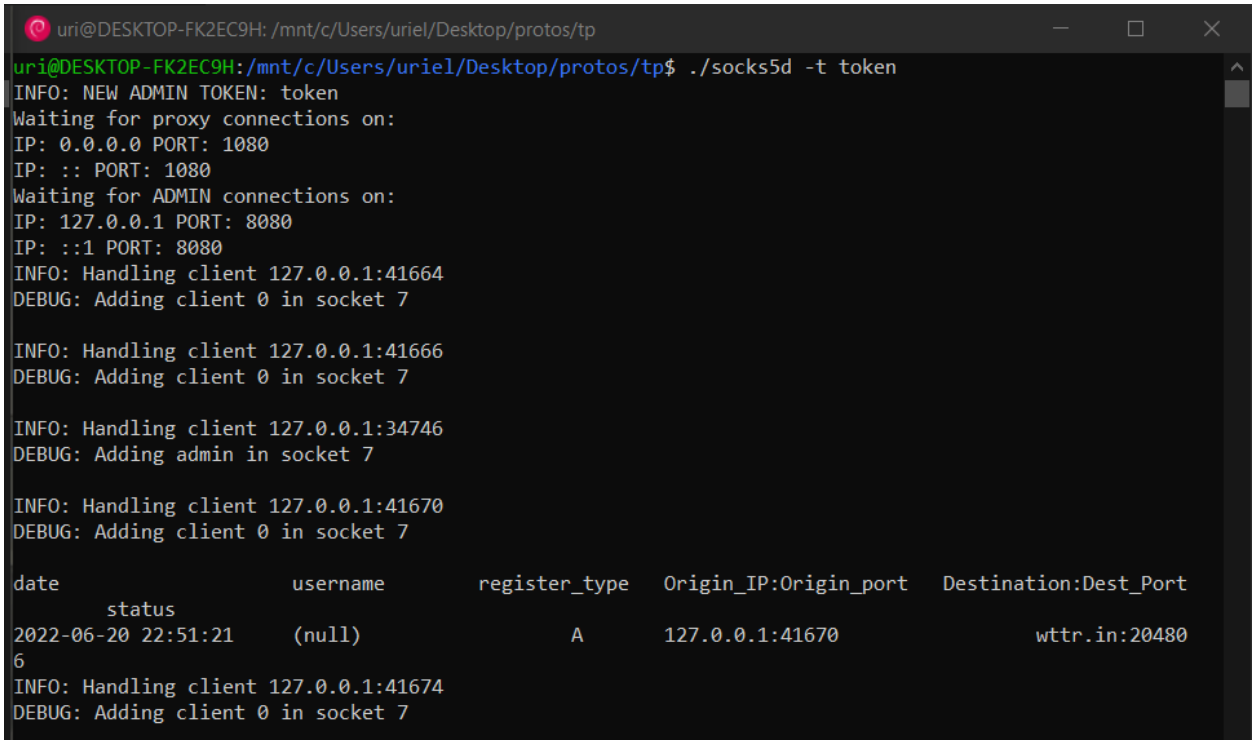
En la primer captura se puede ver cómo se inicializa el servidor con 1 usuario, luego en la segunda captura se consigue esa lista, se la modifica, y se la vuelve a conseguir.

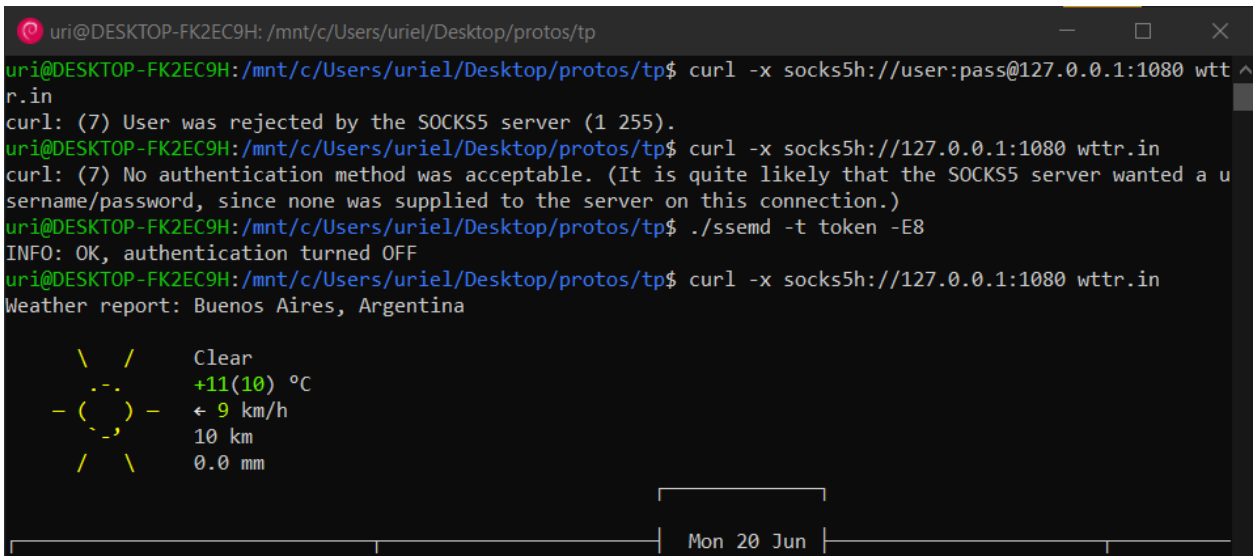
```
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./socks5d -t token -u user:pass
INFO: NEW ADMIN TOKEN: token
Waiting for proxy connections on:
IP: 0.0.0.0 PORT: 1080
IP: :: PORT: 1080
Waiting for ADMIN connections on:
IP: 127.0.0.1 PORT: 8080
IP: ::1 PORT: 8080
INFO: Handling client 127.0.0.1:34722
DEBUG: Adding admin in socket 7

INFO: Handling client 127.0.0.1:34724
DEBUG: Adding admin in socket 7

INFO: Handling client 127.0.0.1:34726
DEBUG: Adding admin in socket 7

INFO: Handling client 127.0.0.1:34728
DEBUG: Adding admin in socket 7

INFO: Handling client 127.0.0.1:34730
DEBUG: Adding admin in socket 7
```

```
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -t token -G4
INFO: List of registered [user:password]:

        user:pass

uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -t token -E5 -d juan:perez
INFO: OK, added new user
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -t token -G4
INFO: List of registered [user:password]:

        user:pass
        juan:perez

uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -t token -E6 -d user:pass
INFO: OK, user removed
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -t token -G4
INFO: List of registered [user:password]:

        juan:perez
```

Editar autenticación:

En la siguiente captura captura se puede ver como se inicializó el Servidor con la autenticación prendida (predeterminadamente, -N para que empieze apagada), y sin registrar ningún usuario. Luego en la siguiente captura se ve cómo 2 clientes son rechazados (uno con user:pass no aceptados y el siguiente sin método de autenticación). Luego el Admin apaga la autenticación, y un cliente puede hacer el pedido sin usuario ni contraseña.





POP3 disector:

Con el disector activado se muestra la password

```
INFO: NEW ADMIN TOKEN: j
Waiting for proxy connections on:
IP: 0.0.0.0 PORT: 1080
IP: :: PORT: 1080
Waiting for ADMIN connections on:
IP: 127.0.0.1 PORT: 8080
IP: ::1 PORT: 8080
date                username        register_type   Origin_IP:Origin_port   Destination:Dest_Port           status
2022-06-21 12:49:43     user            A               127.0.0.1:33090         localhost:110                   0
date                username        register_type   protocol                usr             pwd                     dest_address:dest_port
2022-06-21 12:49:56     user            p               POP3                    tataabancens    lapassword              localhost:110
```

Cambio de direcciones y puertos del servidor, para clientes y admin (-l , -L , -p , -P):

Primero se ven las direcciones y puertos predeterminados, y luego se ve como los 6 valores son modificables (ipv4 e ipv6 de cliente y admin, y ambos puertos)

```
uri@DESKTOP-FK2EC9H: /mnt/c/Users/uriel/Desktop/protos/tp
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./socks5d -t token
INFO: NEW ADMIN TOKEN: token
Waiting for proxy connections on:
IP: 0.0.0.0 PORT: 1080
IP: :: PORT: 1080
Waiting for ADMIN connections on:
IP: 127.0.0.1 PORT: 8080
IP: ::1 PORT: 8080
```

```
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./socks5d -t token -l 127.0.0.1 -l ::1 -p 1081 -L 127.0.0.2 -L :: -P 8081
INFO: NEW ADMIN TOKEN: token
Waiting for proxy connections on:
IP: 127.0.0.1 PORT: 1081
IP: ::1 PORT: 1081
Waiting for ADMIN connections on:
IP: 127.0.0.2 PORT: 8081
IP: :: PORT: 8081
```

Conexiones actuales: Se puede observar cómo en la terminal izquierda se inician muchas conexiones mientras que la derecha pide la cantidad de conexiones abiertas, notando un cambio y una vuelta a 0 cuando se cortan dichas conexiones.

```
bomb: 119
bomb: 120
bomb: 121
bomb: 122
bomb: 123
bomb: 124
bomb: 125
bomb: 126
bomb: 127
bomb: 128
bomb: 129
bomb: 130
bomb: 131
bomb: 132
bomb: 133
bomb: 134
bomb: 135
bomb: 136
bomb: 137
```

```
uri@DESKTOP-FK2EC9H: /mnt/c/Users/uriel/Desktop/protos/tp
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -G2 -t token
INFO: Quantity of current connections:
INFO: number: 0
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -G2 -t token
INFO: Quantity of current connections:
INFO: number: 53
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -G2 -t token
INFO: Quantity of current connections:
INFO: number: 127
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$ ./ssemd -G2 -t token
INFO: Quantity of current connections:
INFO: number: 0
uri@DESKTOP-FK2EC9H:/mnt/c/Users/uriel/Desktop/protos/tp$
```

Conclusiones

Fue un trabajo muy extenso, de los más complicados de la carrera hasta ahora. Pero un lindo desafío. Pudimos entender cómo funciona un servidor no bloqueante y escalable desde la base. Trabajamos con los distintos estados de una conexión y realizamos los parsers necesarios para la lectura de los mensajes. Mejoramos mucho nuestro nivel de programación en C, aunque siempre te sorprende. Tuvimos la oportunidad de desarrollar nuestro propio protocolo y tomar las decisiones de diseño correspondientes, así como también implementar un cliente para el mismo.

Lo relajante de dejar todo en un TP, es que después podés entregar tranquilo.