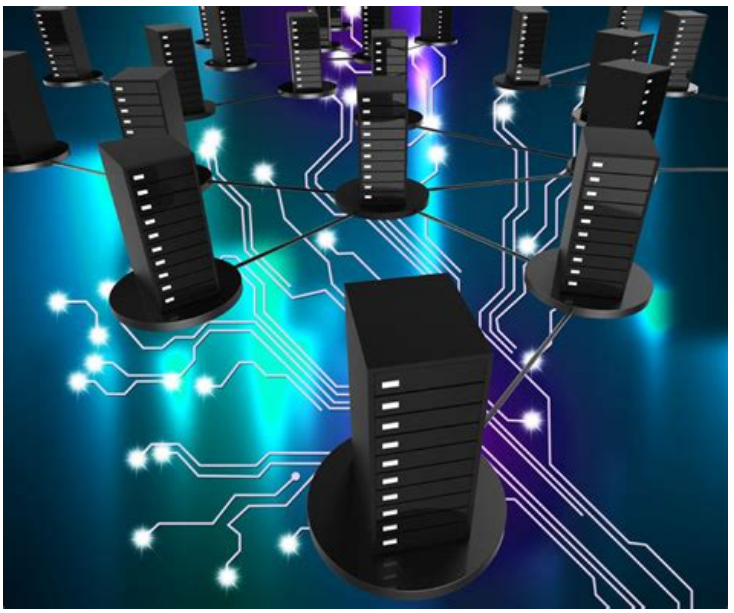
Protocolos de Comunicación
Trabajo Práctico Especial

Informe - TPE Protocolos de Comunicación



Barmasch, Juan Martín (61033),

Bellver, Ezequiel (61268),

Lo Coco, Santiago (61301)

21/07/2022

ITBA

TABLA DE CONTENIDOS

INTRODUCCIÓN

El presente informe detalla las decisiones tomadas por el grupo a la hora de resolver el Trabajo Práctico Especial de la materia 72.07 Protocolos de Comunicación. El mismo consistió en implementar un servidor proxy SOCKSv5 y la creación de un protocolo propio con su respectivo cliente y servidor, cuyo propósito es obtener información del servidor proxy y asimismo proveer una interfaz para su configuración. El objetivo final de este trabajo es "[...] demostrar la habilidad para la programación de aplicaciones cliente/servidor con sockets, la comprensión de estándares de la industria, y la capacidad de diseñar protocolos de aplicación."[1].

DESCRIPCIÓN DETALLADA DE LOS PROTOCOLOS Y APLICACIONES DESARROLLADAS

SOCKSv5

Se implementó un servidor TCP proxy SOCKSv5 no bloqueante mediante el uso de un selector y sockets multiplexadas no bloqueantes, basándonos en el código provisto por la cátedra. Se utiliza una máquina de estados que define las acciones que se realizan en función del estado actual del sistema. Se provee una forma de autenticación mediante usuario y contraseña, en caso de que un cliente no se pueda autenticar, el servidor responde con *status X'FE'* debido a que el *RFC 1929* especifica que en caso de error el *status* debe ser distinto de *X'00'*. También se implementó un disector de contraseñas para el protocolo POP3, el mismo no se limita a leer conexiones que se hacen a un puerto 110.

A la hora de transmitir los datos entre el cliente y el servidor al que se quiera conectar se usa un *buffer* de 4096 bytes, esto es debido a que se probó con 2048 bytes, pero resultaba demasiado lento para consultas reales, por ejemplo *Google Maps*, entre otros. En cambio se encontró que con 4096 la velocidad era más que aceptable. Además, 4096 es el tamaño de página por defecto[2] de Linux, por lo que consideramos que es un valor "*sweet spot"* para transferencia de datos. A su vez, se realizaron pruebas con un buffer de 8192 bytes, pero la mejora de los tiempos de ejecución no resultó proporcional al aumento del tamaño del buffer, por lo que se optó por mantenerlo en 4096 bytes.

Bottler Configuration Protocol (BCP)

Protocolo

El protocolo BCP está montado sobre UDP, es binario, no orientado a sesión y está basado en SOCKSv5. Su propósito es el de proporcionar una forma de configurar un servidor. Cuenta con opciones para manejo de usuarios (ABM), obtención de estadísticas y configuraciones y habilitación/deshabilitación de servicios del servidor.

Este protocolo admite un sistema de autenticación mediante el uso de un *token* de 8 bytes de largo que será enviado en cada *request*. Esto permite verificar un pedido al servidor, lo que puede resultar de interés al tratarse de un protocolo de configuración.

Se tomó la decisión de hacerlo montado sobre UDP por sobre TCP debido a que al tratarse de interacciones pequeñas no se vió la necesidad de tener una conexión persistente. Es decir, esto ahorró la necesidad del *three-way handshake*, lo que provee una baja latencia. Además, siempre se envía información que cabe en un datagrama, evitando el problema de tener que fragmentar la información, con la excepción del listado de usuarios para el cual se decidió dividir en páginas proveyendo una mayor escalabilidad con la cantidad de usuarios. Se envían 5 usuarios por página debido a que el largo de los mismos puede ser de 255 máximo, a esto se le suma el separador (‘\n’). Entonces al mandar 5 usuarios, ocupan a lo sumo 1280 bytes, que es menor que 1500 (MTU de enlace). En caso de recibir menos de 5 usuarios en una página, el último usuario de la misma termina con el separador 2 veces.

Se decidió implementar un protocolo binario para que sea *computer-friendly* y fácil de parsear, pudiendo encargarse el cliente de proveer la información de forma *human-readable*.

Servidor

Se implementó un servidor para el protocolo BCP de forma parecida que para SOCKSv5, no bloqueante, con una máquina de estados y un selector, pero en este caso el protocolo es sobre UDP. Al ser un protocolo de configuración, cuando se recibe un mensaje se envía instantáneamente una respuesta. No teniendo que esperar por una respuesta de otro servidor.

Cliente

Se implementó un cliente bloqueante para el protocolo BCP. Este, al ser ejecutado, recibe opciones por línea de comandos que especifica el comando que se quiere ejecutar. Se encarga de mandar la consulta al servidor y traducir las respuestas que recibe a datos *human-readable*.

PROBLEMAS ENCONTRADOS DURANTE EL DISEÑO Y LA IMPLEMENTACIÓN

Paginación de usuarios

A la hora de cumplir el requerimiento de retornar listas de usuarios del servidor, se optó por implementar una paginación de la misma de forma tal que cada página entre en una trama Ethernet. Esta restricción implicó un problema a la hora de definir cómo dividir dichas páginas. En una primera instancia se pensó en definir las páginas de forma tal que un nombre de usuario pudiera empezar en una página y terminar en otra, indicando que el mismo estaba truncado (de forma similar a como lo hace DNS). Luego de algunas pruebas, se descubrió que, a causa de la implementación que se tenía en el momento, esto representaba un problema en el caso puntual en que el último nombre de usuario terminase justo al final del tamaño máximo de la página. Esto resultaba problemático ya que implicaba la necesidad de predecir si la página anterior estaba siendo ocupada en su totalidad o no para saber si el primer usuario de la página actual era el final del último usuario de la anterior. Se pensó en distintas alternativas para solucionar este problema, pero se optó por deprecar la funcionalidad de usuarios truncados y cambiar a un enfoque de máximo 5 usuarios por página. La justificación de por qué se eligió esa cota superior es la que se dio en la descripción del protocolo. Es importante notar que esto se pudo haber implementado de forma más eficiente dado que no es una restricción propia del protocolo definido, sino de la implementación particular del servidor. Se optó por esta implementación dada su simplicidad y los límites de entrega del proyecto.

Parsers

Al implementar el *password dissector* se utilizó el código de la cátedra. Este nos facilitó el *matcheo* con “user ” y “pass “. Pero al querer volver al estado inicial de los parsers, no se podía. Esto se debió en un principio al escaso conocimiento que se tenía sobre el funcionamiento de los mismos. Se realizaba el *parser_reset* en lugares incorrectos del código.

Getaddrinfo

Se perdió bastante tiempo con la lista de *addrinfo* que devuelve el método getaddrinfo. En un principio no se iteraba por la lista provista, de esta manera en caso de no poder hacer un connect con la 1ra IP provista, fallaba. Luego, se recorría la lista, pero no se contemplaba que el connect estaba implementado de forma no bloqueante, iterando por toda la lista sin poder conectarse a ninguna IP. Finalmente, se dio con la solución utilizada, iterando únicamente si no se lograba conectar (de forma no bloqueante).

Strcpy

En el *server.c* cuando se quiere listar a los usuarios, se realiza un strcpy del nombre del usuario al buffer de salida. Esto se hacía en un *while* que incrementaba la variable *i*. Al hacerlo en el *strcpy* de la siguiente manera:

```
i = start;
while (i < start + PAGE_SIZE && i < args->nusers) {
   len += args->users[i].ulen;
   strcpy(buff + len - args->users[i].ulen, args->users[i++].name);
   buff[len - 1] = '\n';
}
```

no funcionaba debido a que incrementaba *i* antes de ejecutar la acción. Luego de varios intentos se probó:

```
i = start;
while (i < start + PAGE_SIZE && i < args->nusers) {
    strcpy(buff + len, args->users[i].name);
    len += args->users[i++].ulen;
    buff[len - 1] = '\n';
}
```

que soluciona el problema y al mismo tiempo simplifica el código.

Devolución de códigos de error

A la hora de devolver un código de error al cliente, resultó de dificultad decidir en qué parte del código hacerlo. Se debieron implementar estados que se encargan de retornar el código y luego cortar la ejecución del *socket* activo. De esta forma, se provee al cliente una noción de qué sucedió sin cortar la conexión directamente.

LIMITACIONES DE LA APLICACIÓN

Debido a que se usó la función *pselect*, que solo admite hasta 1024 *file descriptors*, se impusó una restricción de 500 conexiones simultáneas.

Además, es preciso notar que solo se provee un token válido para poder acceder al servicio de configuración, siendo que si se quiere utilizar un token distinto se debe reiniciar el servidor y cambiar la variable de entorno. Una implementación posible es la de contar con más de un token válido o que pueda ser auto generado.

POSIBLES EXTENSIONES

Se podría, para no tener la restricción mencionada en la sección anterior de los *file descriptors*, utilizar *epoll*.

En el caso del servidor de configuración se puede proveer un mejor listado de usuarios, aprovechando en su totalidad el tamaño de la página impuesto, de esta forma en situaciones donde se cuenta con muchos usuarios no se requieren muchas consultas al servidor para poder listarlos a todos.

Por otro lado, en el cliente del protocolo de configuración, se podría haber implementado las siguientes funcionalidades: reinicio del servidor, cambio del tamaño del buffer, cambio del timeout del select, obtención de los usuarios y contraseñas obtenidos por el *password dissector*, obtener la última conexión de un usuario, entre otras.

CONCLUSIÓN

El trabajo en cuestión requirió una vasta integración de conceptos vistos en la materia, mediante el cual se consiguió una mejor comprensión del funcionamiento de un servidor, un cliente y de la implementación de un protocolo propio. Además, debido a la elección de realizar nuestro protocolo sobre UDP se pudo trabajar con TCP y UDP, permitiendo abarcar una mayor cantidad de los conceptos vistos en la materia.

Se precisó tomar decisiones en cada momento del desarrollo del proyecto, donde se necesitó tener en consideración los pros y los contras de cada opción, que debido a nuestra forma de trabajar, debía llevar a un mutuo acuerdo entre los integrantes del equipo a la hora de decidir cuáles eran los pasos a seguir.

EJEMPLOS DE PRUEBA

Se realizaron pruebas del servidor Proxy con los siguientes clientes: *curl*, *netcat*, *Chromium* y *Mozilla Firefox*. En los web browsers mencionados se ingresó a diversos sitios que permitieran demostrar la eficiencia del servidor implementado, tales como http2demo.io y Google Street View.

A su vez se utilizó una extensión de Chrome que permite recargar múltiples pestañas en simultáneo, logrando así llegar a tener 58 conexiones concurrentes transfiriendo datos (567895441 bytes).

```
› ./client 1148926835748244254 -L 127.0.0.1 -P 8080 -m
Connecting to 127.0.0.1:8080

Metrics:
        Total connections: 2614
        Current connections: 49
        Total bytes transferred: 567895441
```

Luego, se realizó una prueba de stress que consistía en levantar un servidor http y realizar un programa *bash* que genera hijos (realiza forks) donde cada uno se conecta a través del proxy y descarga un archivo de 120MB. Llegando hasta 509 conexiones concurrentes:

```
Connecting to 127.0.0.1:8080

Metrics:
        Total connections: 830
        Current connections: 509
        Total bytes transferred: 37689556090
```

Analicemos un poco este número: sabemos que cada conexión tendrá asociado 2 files descriptors y el server, por su lado, tendrá ocupados 6 files descriptors (stderr, stdout y los 4 que se utilizan para las conexiones TCP y UDP pues se aceptan tanto IPv4 como IPv6). Sumando todos estos valores quedaría un total de $2 * 509 + 6 = 1024$. Dentro de este valor se encuentran también las conexiones del cliente (aunque se considera como 1 debido a que se ejecuta en un *for*).

Se muestran algunos GET al servidor:

```
127.0.0.1 - - [20/Jun/2022 20:27:15] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:15] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:15] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:15] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:18] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:18] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:18] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:18] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:18] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:18] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:25] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
127.0.0.1 - - [20/Jun/2022 20:27:25] "GET /Downloads/brave-bin-1:1.37.116-1-x86_64.p
kg.tar.zst HTTP/1.1" 200 -
```

A continuación, adjuntamos los pasos seguidos (previo a esto se inicializó el server http en 127.0.0.1:8000 y se desactivó la autenticación mediante usuario/contraseña):

```
#!/bin/bash
for i in {1..1000}; do
   curl -x socks5://127.0.0.1:1080
http://127.0.0.1:8000/Downloads/brave-bin-1:1.37.116-1-x86_64.pkg.tar.zst
&> /dev/null &
done
```

Además, para poder ver las conexiones concurrentes:

```
#!/bin/bash
for i in {1..1000}; do
   ./client 1148926835748244254 -L 127.0.0.1 -P 8080 -m
done
```

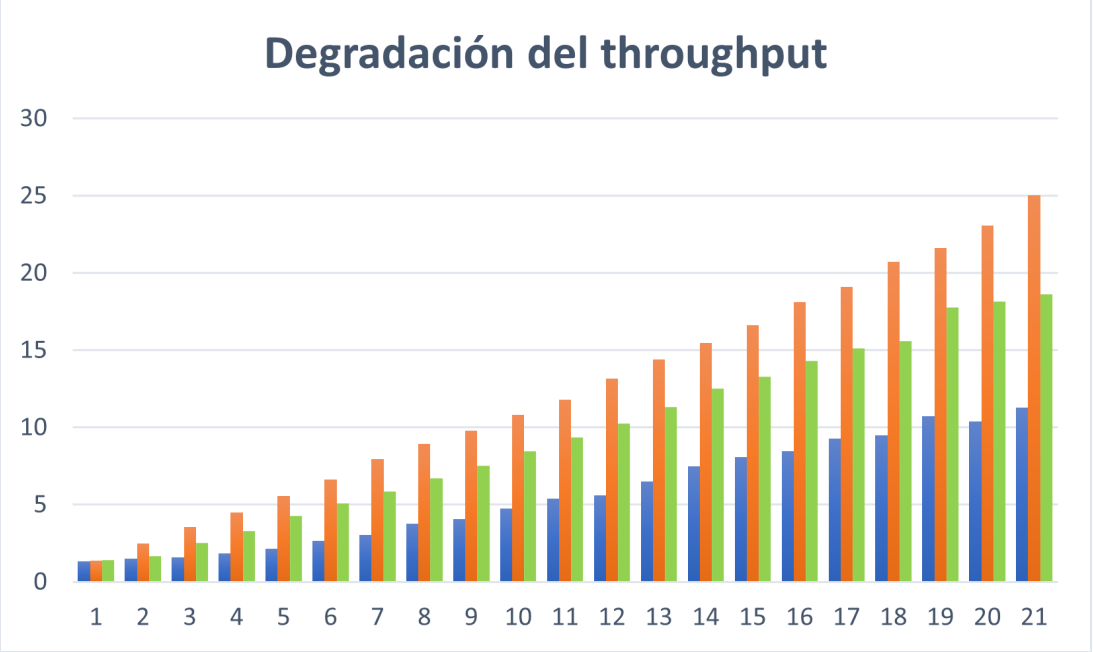
Por otro lado, se analizó la degradación del *throughput* en función del número de conexiones concurrentes. Para esto se construyó el siguiente código:

```
#!/bin/bash
for i in {1..20}; do
   for j in $(seq 0 $i); do
      (time curl -x socks5://127.0.0.1:1080 -s
```

```
http://127.0.0.1:8000/Downloads/archlinux-2022.03.01-x86_64.iso |
md5sum) &
    done
    for j in $(seq 0 $i); do
        wait
    done
    printf "\n\n\n\n\n"
done
```

En el código anterior se hacen dos ciclos. En el ciclo interno se itera desde 1 hasta i (variable del ciclo externo) y se calcula el tiempo que tarda un curl al servidor *HTTP* (el mismo que el del ejemplo anterior) de un archivo que pesa $801MB$ usando el proxy y, además, calculando el hash *MD5* (para ver que no haya corrupción de datos). Aquí hacemos, como en el ejemplo anterior, forks para el ciclo interno (pues queremos que sean concurrentes) y luego otro ciclo en el cual se espera con un *wait* que todos ellos terminen para pasar al siguiente ciclo (externo).

Luego, se realizó la misma prueba pero sin usar el proxy.



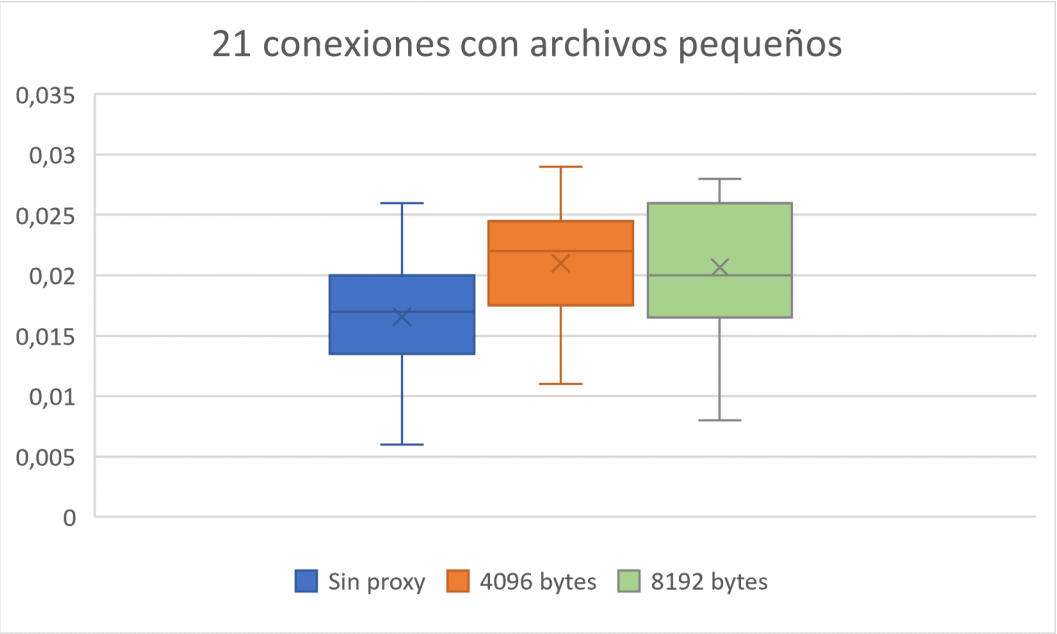
Este gráfico muestra la degradación del *throughput* (tiempo vs. cantidad de conexiones concurrentes) con el proxy y un buffer de 4096 bytes (naranja), con el proxy y un buffer de 8192 bytes (verde) y sin el proxy (azul). Se puede ver que, en promedio, por cada conexión concurrente que se agrega, el tiempo con el proxy se ve afectado en aproximadamente un poco más de un segundo, exactamente $1,1824s$ (contra $0,49785s$ de

promedio sin el proxy). Este resultado está dentro de lo esperable ya que el proxy divide el tiempo de procesamiento para encargarse de cada conexión, cuantas más de las mismas haya.

Se muestra la salida del último ciclo (recortada):

```
real    0m10.718s                       real    0m24.582s
user    0m1.406s                        user    0m1.619s
sys     0m0.737s                        sys     0m1.056s
74028867e3a1f02ed4d252fe4674beb1  -     74028867e3a1f02ed4d252fe4674beb1  -

real    0m11.183s                       real    0m24.588s
user    0m1.411s                        user    0m1.574s
sys     0m0.742s                        sys     0m1.118s
74028867e3a1f02ed4d252fe4674beb1  -     74028867e3a1f02ed4d252fe4674beb1  -

real    0m11.192s                       real    0m24.587s
user    0m1.485s                        user    0m1.635s
sys     0m0.659s                        sys     0m1.042s
74028867e3a1f02ed4d252fe4674beb1  -     74028867e3a1f02ed4d252fe4674beb1  -

real    0m11.221s                       real    0m24.594s
user    0m1.444s                        user    0m1.580s
sys     0m0.702s                        sys     0m1.092s
74028867e3a1f02ed4d252fe4674beb1  -     74028867e3a1f02ed4d252fe4674beb1  -

real    0m11.226s                       real    0m24.943s
user    0m1.471s                        user    0m1.615s
sys     0m0.674s                        sys     0m1.052s
74028867e3a1f02ed4d252fe4674beb1  -     74028867e3a1f02ed4d252fe4674beb1  -

real    0m11.236s                       74028867e3a1f02ed4d252fe4674beb1  -
user    0m1.435s                        74028867e3a1f02ed4d252fe4674beb1  -
sys     0m0.717s                        real    0m25.014s
74028867e3a1f02ed4d252fe4674beb1  -     user    0m1.590s
                                        sys     0m1.070s
real    0m11.247s
74028867e3a1f02ed4d252fe4674beb1  -     real    0m25.012s
user    0m1.423s                        user    0m1.553s
sys     0m0.726s                        sys     0m1.103s

real    0m11.234s                       real    0m25.010s
user    0m1.403s                        user    0m1.641s
sys     0m0.748s                        sys     0m1.015s
74028867e3a1f02ed4d252fe4674beb1  -     74028867e3a1f02ed4d252fe4674beb1  -
                                        74028867e3a1f02ed4d252fe4674beb1  -
real    0m11.271s
user    0m1.426s                        real    0m25.020s
sys     0m0.718s                        user    0m1.639s
74028867e3a1f02ed4d252fe4674beb1  -     sys     0m1.024s

real    0m11.297s                       real    0m25.020s
user    0m1.493s                        user    0m1.616s
sys     0m0.648s                        sys     0m1.050s
```

Además, se realizó una última prueba con un archivo más pequeño (300 bytes).



Se debe notar que, en base a los gráficos anteriores, no justifica el uso de un buffer del doble de tamaño debido al overhead de memoria que esto conlleva (alocar el doble de memoria).

GUÍA DE INSTALACIÓN

Esta se encuentra detallada en el archivo *README.md* en la raíz del proyecto.

INSTRUCCIONES PARA LA CONFIGURACIÓN

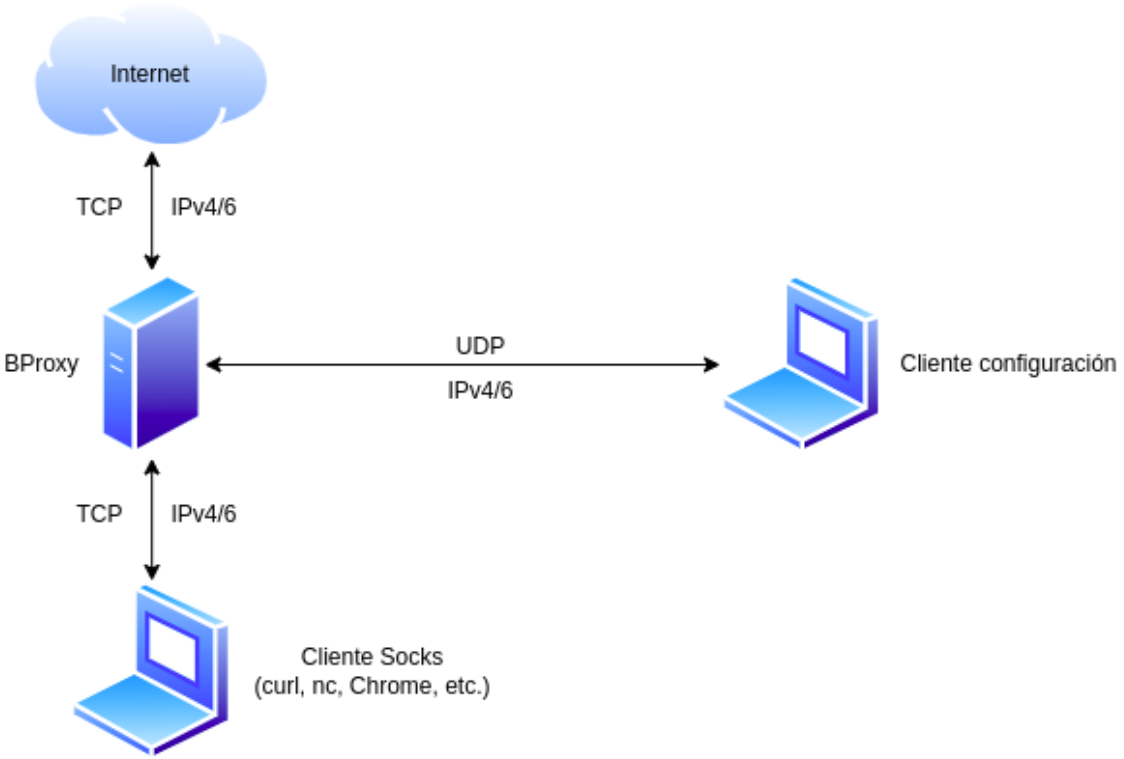
A la hora de correr el cliente, en caso de querer hacer una consulta al servidor, el mismo acepta

```
./client [TOKEN] -L <dirección IP> -P <puerto> -"comando" <parámetros>
```

en simultáneo. No acepta más de un comando por ejecución. En caso de recibir más de uno, se correrá únicamente el 1ro que se reciba. Para ejecutar un comando en el servidor es obligatorio proveer un token. En cambio, para correr la ayuda o la información de la versión del cliente ("-h" y "-v" respectivamente) no es necesario.

EJEMPLOS DE CONFIGURACIÓN Y MONITOREO

Se proveen ejemplos en el archivo *README.md* en la raíz del proyecto.



REFERENCIAS

[1]: Enunciado del trabajo práctico.

[2]: Manual de *getconf*: https://linux.die.net/man/1/getconf