Protocolos de Comunicación

TPE: Servidor POP3

Profesor: Juan Francisco Codagnone, Sebastian Kulesz

Alumnos: Mauro Leandro Báez, Franco David Rupnik, Matías Manzur y Federico Shih

1. Índice

2. Descripción de Trabajo Práctico

El trabajo práctico se trata de la implementación de un servidor POP3 en acorde al RFC 1939, creación e implementación de un protocolo de monitoreo y configuración del servidor implementado, y la implementación de un cliente que utiliza el protocolo de comunicación previamente mencionado.

Implementamos un servidor de POP3 no bloqueante, que provee y acepta conexiones POP3 en el puerto 1110, un protocolo de comunicación denominado PEEP para configurar el servidor, y un cliente el cual actúa de intermediario entre el usuario y la moderación/estadísticas del servidor POP3.

También implementamos un mecanismo de transformación, que mediante un programa externo realiza la transformación de mails del usuario automáticamente. Estos programas tienen que recibir por stdin el mail y enviar por stdout el resultado de transformación.

En un principio, incluso sin un cliente se podría utilizar correctamente PEEP con todas sus funcionalidades. El problema con esto radica en que el usuario debería conocer a la perfección cómo el servidor espera los comandos y los códigos de error, los cuales no fueron diseñados para ser intuitivos al ojo humano, sino que apuntaban a hacerle la vida más fácil a quien implementa el cliente. Por ejemplo, para añadir usuarios se debe seguir la notación “+a <nombre> <contraseña>”, y de ya haber un usuario con ese nombre, el mensaje de error sería simplemente “-3”. Por otra parte, por medio del cliente, el usuario podría escribir algo como “add user <nombre> <contraseña>” y de haber un error, el cliente le informaría “Username already in use” o algo por el estilo.

Este servidor recibe por argumento y/o por configuración externa: la ubicación de correos, los usuarios de POP3, el usuario de PEEP, el programa de transformación, los puertos a utilizar para escuchar pedidos a conexión POP3 o PEEP, la cantidad de conexiones máximas y timeout de usuario.

Nuestro servidor implementa autenticación por USER PASS, CAPA y PIPELINING.

PEEP es un protocolo orientado a conexión basado sobre TCP. Utiliza un esquema de comando respuesta, permitiendo operaciones simples para comunicación simple y mayor extensibilidad. Para una definición más extensiva de este protocolo ver el archivo adjunto a modo de RFC del protocolo.

El trabajo es desarrollado utilizando C11, con la versión de POSIX C 200112L.

3. Problemas encontrados en diseño e implementación

El primer problema con el que nos encontramos (debido a que elegimos de entrada hacer una implementación no bloqueante) fue la elección de qué implementación de selector utilizar para monitorear los sockets (file descriptors). Vimos las implementaciones de select, pselect, el selector dado por la cátedra y finalmente poll. Tomamos la decisión de utilizar poll debido a que es más cómodo de implementar pues sólo pide una lista de los file descriptors a observar (y si queremos escribir o leer), mientras que por ejemplo el select necesita tres mapas de bits que pueden llegar a desperdiciar mucho espacio si hay un sólo socket con un número de file descriptor muy alto.

Como el selector puede despertar al proceso cuando sólo está listo para leer de un cliente pero no escribir, al procesar un comando decidimos escribir la respuesta que debe ser enviada al cliente en un buffer intermedio de escritura. Luego, cuando el selector avise que se puede escribir al cliente, se leerá de este buffer intermedio la respuesta a enviar.

El siguiente problema que encontramos fue cómo manejar la respuesta de ciertos comandos (como LIST o RETR) cuando las mismas se extendían del máximo que se podía escribir en el buffer intermedio. Como solución, mantuvimos el estado de la conexión de un cliente y agregamos la posibilidad de dejar un comando pendiente en el estado del cliente.

De esta manera, fue posible que el comando guarde dónde se quedó en su procesamiento y continúe cuando el buffer vuelva a tener espacio. Cabe aclarar que habrá a lo sumo un comando pendiente a la vez por cada cliente, es decir que mientras haya un comando pendiente, no se procesan nuevos comandos del cliente. Aun así, si estos ya se habían leído, se mantienen guardados en el estado del parser de cada cliente, listos para ser procesados en cuanto termine el comando pendiente.

Otro problema fue cómo implementar la transformación de mensajes. Teníamos la idea de cómo realizarlo (pipe, fork, dup2, execve) pero iba a cambiar mucho el código. Finalmente encontramos una solución utilizando POPEN, que realiza el fork-execve automáticamente con un pipe unidireccional (en nuestro caso, read). Esto nos permite utilizar CAT por default como método de lectura de archivo, cuya respuesta pasaría por stdin a cualquier programa que se le indique. Debido a nuestra implementación de RETR, pudimos realizar la transformación de file descriptor de archivo a file descriptor de stream de una forma bastante transparente. Debido a que no estamos monitoreando el file descriptor de lectura, hacemos que sea un fd no bloqueante utilizando FNCTL permitiendo que no bloquee por IO el resto del servidor.

Además, para establecer un máximo dinámico de conexiones simultáneas (que puede ser establecido a través de PEEP, aunque no puede ser mayor a 998), decidimos dejar de leer nuevos pedidos de conexión de los sockets pasivos mientras

que la cantidad de conexiones actual sea mayor o igual que el máximo. De esta manera, se encolan los pedidos de conexión en los sockets pasivos, y cuando vuelva a haber lugar, se leerán de los mismos los pedidos que hayan quedado encolados.

Por otro lado, otro problema encontrado fue manejar las liberaciones de memoria cuando un cliente terminaba su sesión ya sea de la forma adecuada (con el comando QUIT), por un error inesperado o por una señal(Kill o Ctrl+C). En un principio, se delegaba la liberación de los recursos usados a los handlers de los mismos clientes (ya sea de POP3 o PEEP). Esto funcionaba correctamente hasta que vimos el caso de que hubiera una señal para matar al proceso del lado del cliente. Esta última señal, no se maneja del lado del handler sino que lo hace el selector y necesitábamos saber qué tipo de socket era para poder llamar a la función de liberación del estado del cliente correspondiente. Para eso, agregamos al estado del socket un puntero a función que se encargaba de liberar el estado del cliente.

Cabe aclarar que para el uso de free, debido a que se extiende mucho si estamos chequeando si se asigna memoria a cada parte distinta del estado del cliente, hacemos un calloc para darle memoria a la estructura y siempre llamamos al free de todos los punteros de la estructura. Esto lo vimos adecuado porque free no realiza ninguna operación si el puntero es NULL, entonces no hay problema mientras siempre que se inicialice la estructura con calloc.

Realizando testing de stress, utilizamos Javascript para realizar conexiones concurrentes. Sin embargo, vimos que había un problema al estar realizando conexiones simultáneas seguidas. Algunas conexiones no son aceptadas, y recien son detectados por Poll en caso de que se corte la conexión (matando el script de testing). No sabemos por qué sucede esto, y nos obliga a dar un pequeño timeout entre cada conexión para el script. De esta manera lo forzamos a que tenga un sleep (en el script de testing) para poder demostrar que las conexiones concurrentes sucedían.

4. Limitaciones

Una limitación con la cual nos encontramos al hacer el protocolo de moderación/control (PEEP) fue el hecho de que todas las estadísticas que el administrador puede visualizar son exclusivamente volátiles. Es decir que una vez que se apaga el servidor, dichos datos son perdidos. Esto estaba contemplado y permitido en la consigna, pero no deja de ser una limitación.

Otra limitación viene dada por el desaprovechamiento de procesadores multicore, ya que todo nuestro servidor funciona en un core, como fue comentado en su momento por la cátedra.

Una limitación más para mencionar es que la cantidad de sockets máxima es estática y está limitada a 1000 sockets. Decidimos limitar la cantidad máxima debido a que el poll no funciona con una cantidad ilimitada de file descriptors para monitorear (es una limitación del sistema operativo). Además para agilizar el uso de una gran cantidad de sockets, se instancia un array estático de 1000 lugares, los

cuales se usan y liberan a medida que se usan los sockets. Ahora, en lo que concierne la elección de la cantidad de lugares, debido a que la consigna pide mínimamente 500 clientes simultáneos, decidimos que duplicar este número era razonable.

Finalmente otra limitación se puede ver en la indexación y lectura de mails que llegan de forma dinámica. Esto es debido a que no tenemos acceso al manejo de la llegada de mails a los directorios, pues no tenemos comunicación con un servidor SMTP. Al momento que un usuario inicia una sesión, se indexan los mails que están presentes en ese momento en el directorio y son los que aparecerán en el retr y en el list. Además, suponemos que no habrá alteraciones a los mails que estén presentes en el directorio, debido a que sólo se podrían eliminar usando una sesión de POP3, la cual ya estará en proceso al momento de leer un mail o listar los mismos.

Una pequeña consideración debe hacerse, los mails obtenidos mediante RETR, difieren de los obtenidos usando Dovecot en el caso en el cual dicho mail no esté usando CRLF como separador de líneas. Dovecot, reemplaza todo LF por CRLF, mientras que nosotros simplemente agregamos un CRLF al final del mail. Luego de esto, agregamos la línea final con ".CRLF", por lo que el mail diferirá del original pues tendrá un CRLF de más al final (Esto sólo ocurre con mails no terminados en CRLF, para los mails que sí lo hacen el mail se mantiene inalterado).

Qué hacer cuando un mail finaliza en CRLF fue debatido en clase de manera álgida. Algunos argumentaban que tenía sentido no agregar otro CRLF al mail si este ya terminaba con uno. Nosotros coincidimos con esta escuela de pensamiento.

5. Posibles Extensiones

Una posible extensión puede ser hacer que las estadísticas no sean volátiles, escribiendo los datos en un archivo al cerrar el servidor. O también que se escriban a modo de logs cada cierto tiempo, para luego poder modelizar como fue creciendo cada métrica a lo largo del tiempo.

Podríamos aumentar la complejidad del proyecto para que se utilicen múltiples cores, habría que tener mucho cuidado para que ninguno de los procesos termine bloqueado y que no haya problemas de concurrencia.

También se podrían implementar más comandos/extensiones de POP3, lo cual aumentaría sin duda la complejidad del proyecto. Un ejemplo de esto sería usar la forma de autenticación APOP, que está presente como otra forma de iniciar sesión. Otra cosa que se podría implementar es el manejo de expiraciones para los mails, para que se borren de forma periódica luego de ser leídos.

6. Conclusiones

El desarrollo del servidor POP3 nos dió la oportunidad de afianzar nuestro conocimiento en programación con sockets, procesos bloqueantes y selectores. Pudimos aplicar nuestro conocimiento sobre TCP para realizar lecturas y escrituras no bloqueantes. Aprendimos las mejores prácticas para diseñar un protocolo de comunicación y sus dificultades a la hora de diseñar estándares para que sean retrocompatibles y extensibles.

Pudimos crear nuestro propio servidor TCP con las configuraciones que nosotros pensamos aptos para su funcionamiento. Mediante Logging y Métricas, podemos obtener información de funcionamiento del servidor. También mediante nuestro protocolo PEEP podemos cambiar el funcionamiento del servidor en tiempo real, como por ejemplo el tiempo de timeout, el Maildir o los usuarios mismos.

Pudimos realizar testing de nuestro servidor, forzándolo a que funcione en casos más extremos de conexión, aceptando casos borde y fallos del protocolo.

7. Ejemplos de prueba

7.1 Máximo de Conexiones Concurrentes

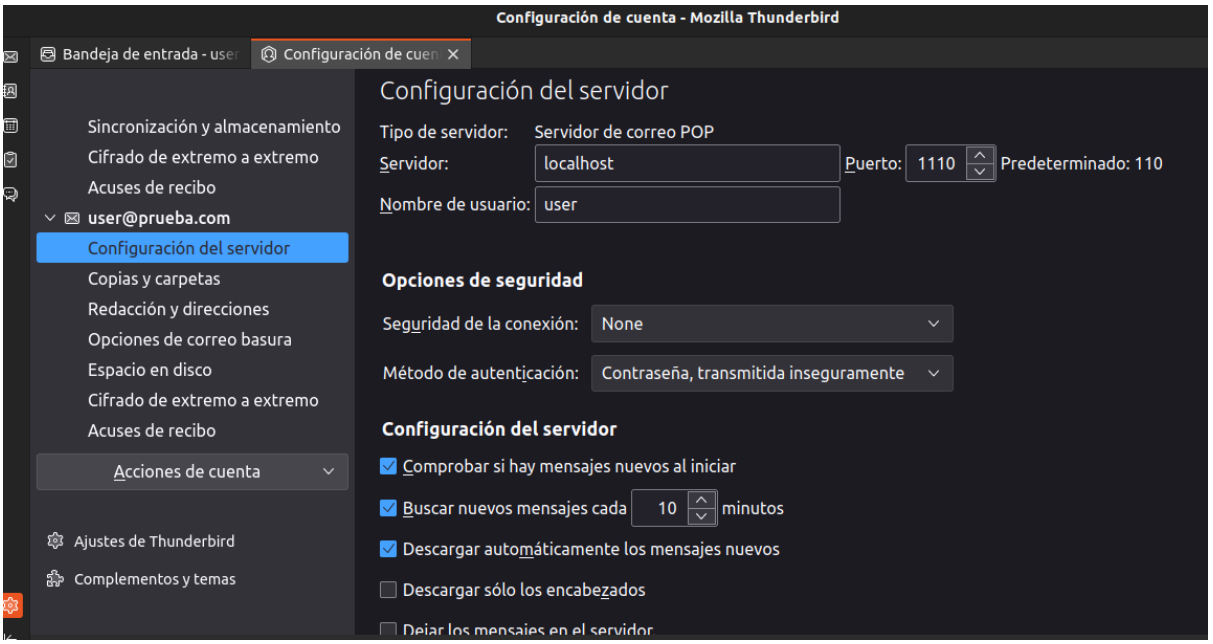
Utilizando el script de stress testing, podemos ver la cantidad de conexiones concurrentes máximas que se pueden realizar. Corriendo npm start con 3000, podemos ver con peep client que la cantidad de conexiones no supera 998.

```
26.48831644281122
^C
shift@shift-system:~/Repositories/pop3-server/stress-testing$ npm start 3000

> stress-testing@1.0.0 start
> node index.js 3000
```

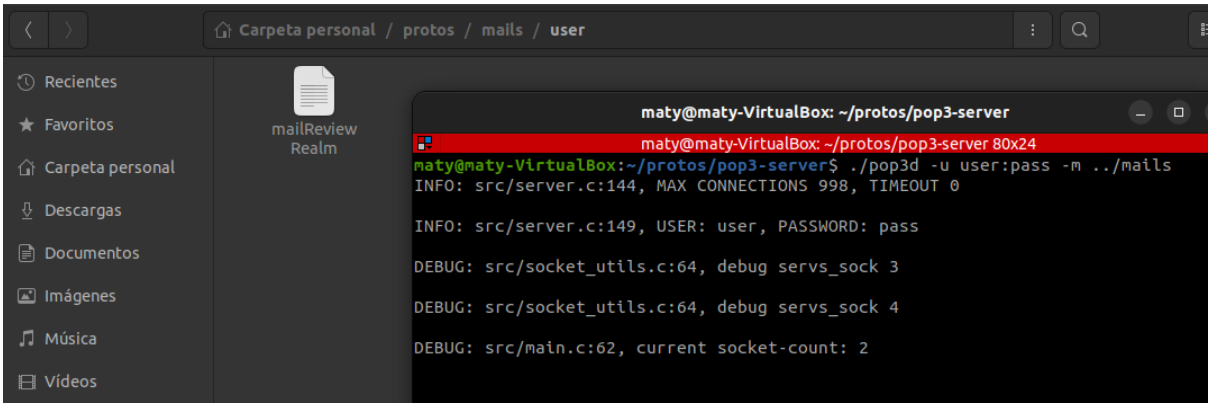
```
OK! 0
current connections amount
OK! 0
current connections amount
OK! 997
current connections amount
OK! 86
current connections amount
OK! 997
current connections amount
OK! 997
```

7.2 Prueba de uso de MUA Thunderbird

Se agregó una cuenta con la siguiente configuración al Mozilla Thunderbird



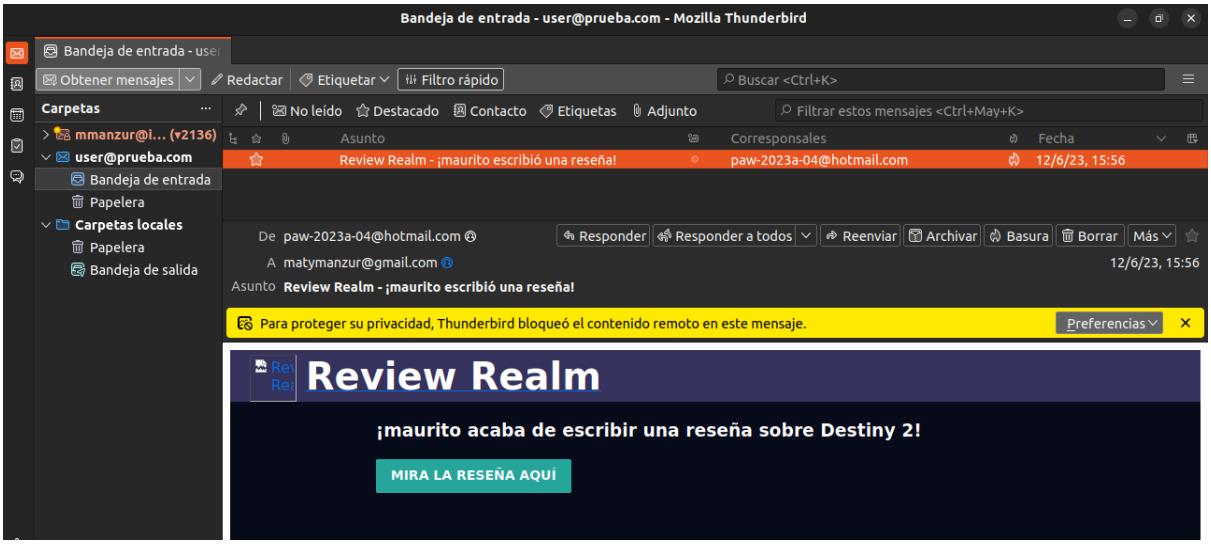
Corremos el servidor pop3 indicando el path donde colocamos un mail del usuario “user” para que vea:



Luego en el Thunderbird, le pedimos que obtenga los mensajes y vemos que aparece el mail que habíamos colocado en el directorio. Y vemos a través de los logs del servidor las siguientes interacciones del MUA:

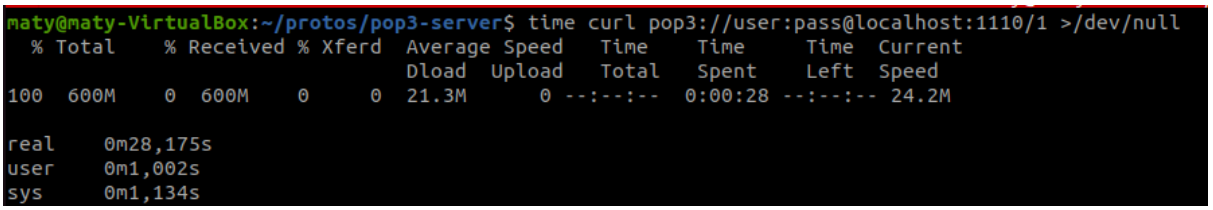
```
DEBUG: src/pop3_utils.c:169, accepted client socket: 2
DEBUG: src/command_utils.c:41, Writing +OK POP3 preparado <pampero.itba.edu.ar> to
socket
DEBUG: src/pop3_utils.c:114, Command CAPA received with args (null) and (null)
DEBUG: src/command_utils.c:41, Writing +OK
USER
PIPELINING
.
```

```
 to socket
DEBUG: src/pop3_utils.c:114, Command USER received with args user and (null)
DEBUG: src/command_utils.c:41, Writing +OK ahora pone la PASS :)
 to socket
DEBUG: src/pop3_utils.c:114, Command PASS received with args pass and (null)
Directory: ../mails/user
INFO: src/directories.c:33, DIRP is NULL 0
INFO: src/pop3_utils.c:405, retrieved emails: 1
DEBUG: src/command_utils.c:41, Writing +OK listo el pollo :)
 to socket
DEBUG: src/pop3_utils.c:114, Command STAT received with args (null) and (null)
DEBUG: src/command_utils.c:41, Writing +OK 1 12247
 to socket
DEBUG: src/pop3_utils.c:114, Command LIST received with args (null) and (null)
DEBUG: src/command_utils.c:41, Writing +OK 1 messages (12247 octets)
1 12247
.
 to socket
DEBUG: src/pop3_utils.c:134, Invalid command UIDL received
DEBUG: src/command_utils.c:41, Writing -ERR invalid command
 to socket
DEBUG: src/pop3_utils.c:114, Command RETR received with args 1 and (null)
DEBUG: src/pop3_utils.c:114, Command DELE received with args 1 and (null)
DEBUG: src/command_utils.c:41, Writing +OK message 1 borrado
 to socket
DEBUG: src/pop3_utils.c:114, Command QUIT received with args (null) and (null)
INFO: src/pop3_utils.c:450, Deleted email ../mails/user/mailReviewRealm
DEBUG: src/command_utils.c:41, Writing +OK See you next time (0 messages left)
 to socket
```

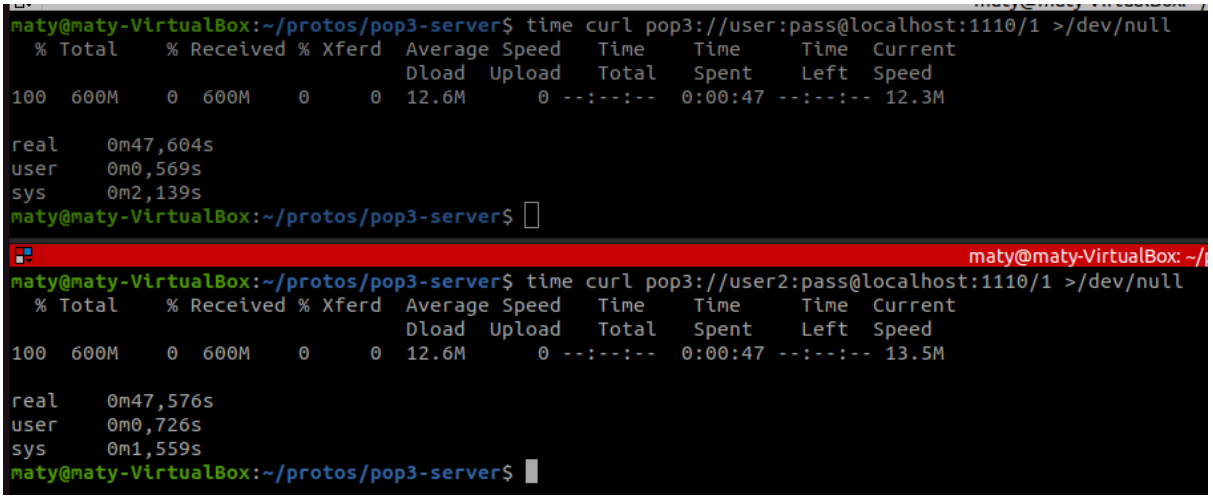


7.3 Ejecuciones de RETR en simultáneo

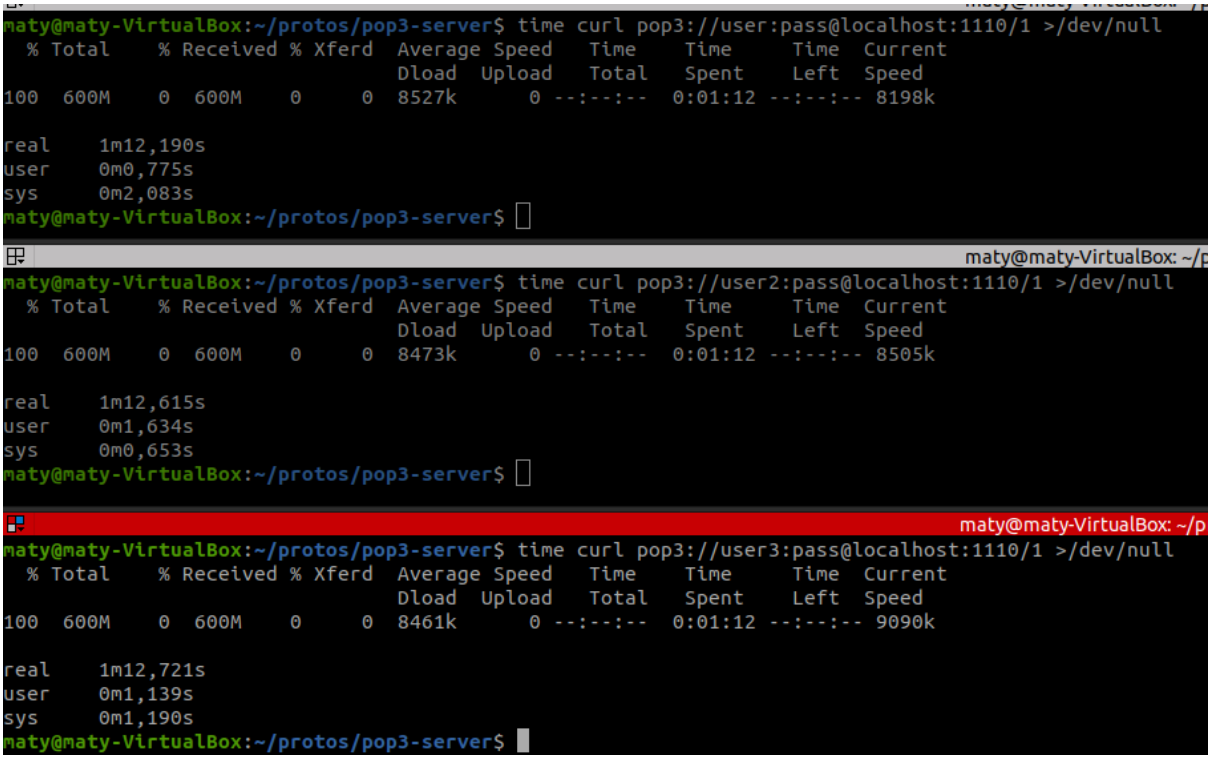
A continuación se ve la pérdida lineal de velocidad a medida que la cantidad de usuarios aumenta. Para explicitarlo, se crearon 4 usuarios, en sus casillas todos los usuarios cuentan con el mismo mail de 600 MB, y se corrió el comando que se muestra en las imágenes de 4 maneras distintas, con 1 usuario, con 2, con 3 y finalmente con 4. Podemos ver la diferencia de velocidad y de tiempo final.
Con 1 Usuario:



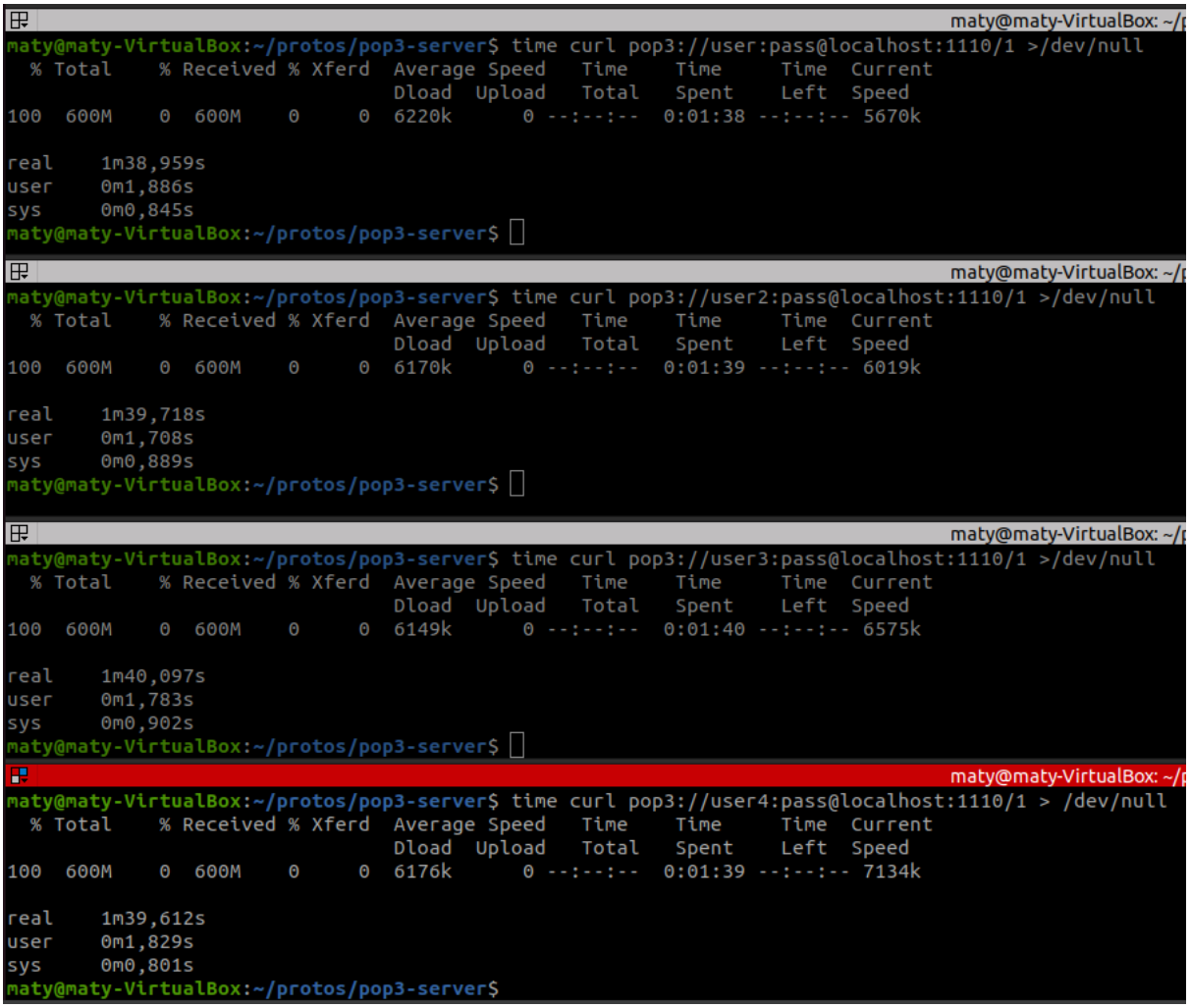
Con 2 Usuarios:



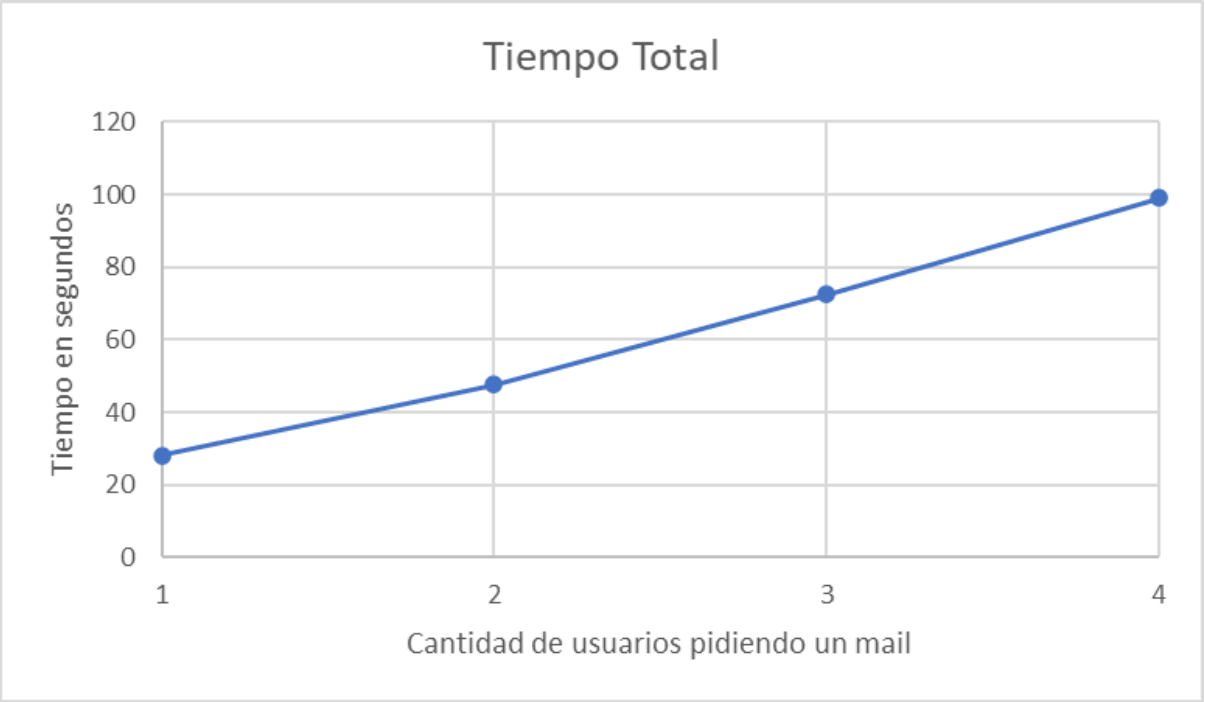
Con 3 Usuarios:

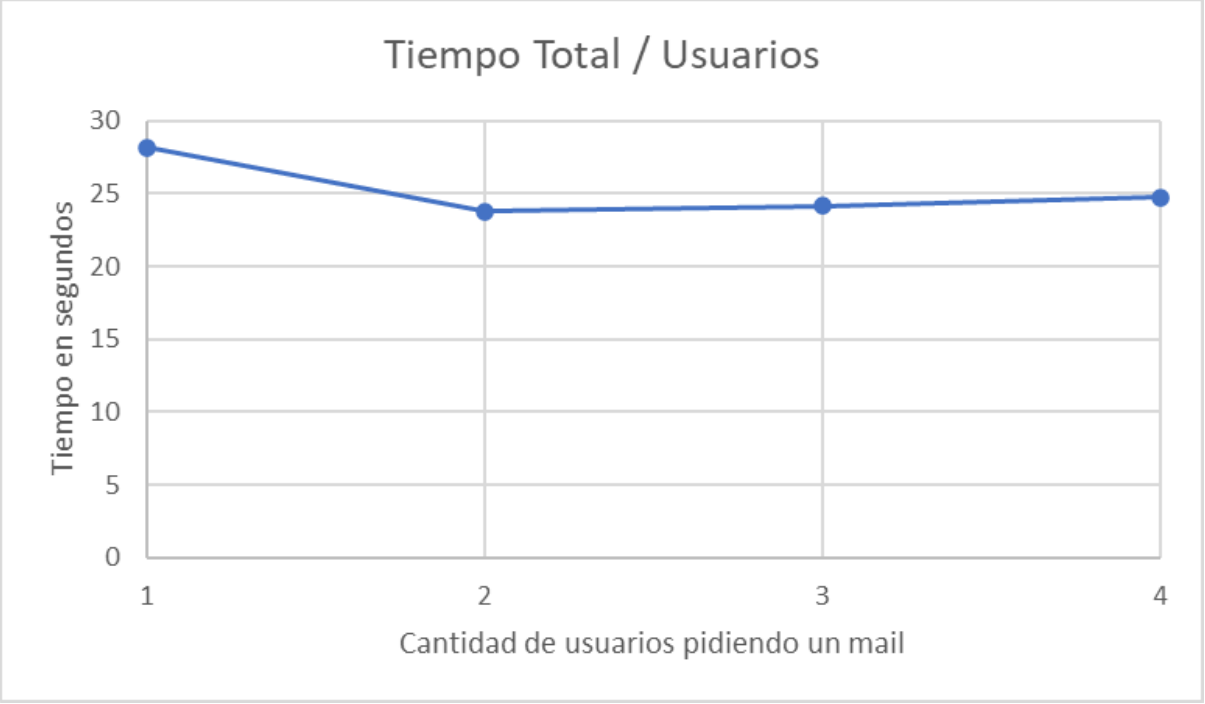


Con 4 Usuarios:



A continuación vemos dos gráficas. La primera muestra el tiempo que tarda el servidor en procesar las solicitudes de los N usuarios. En la misma, vemos que el crecimiento del tiempo es lineal a mayor cantidad de usuarios. De hecho, en la segunda gráfica se muestra el tiempo por usuario (es decir, el tiempo total que tarda el servidor en procesar la solicitud de los N usuarios, dividido la cantidad de usuarios N) en función de la cantidad de usuarios, y vemos que es considerablemente constante, lo que reafirma la linealidad de esta relación.

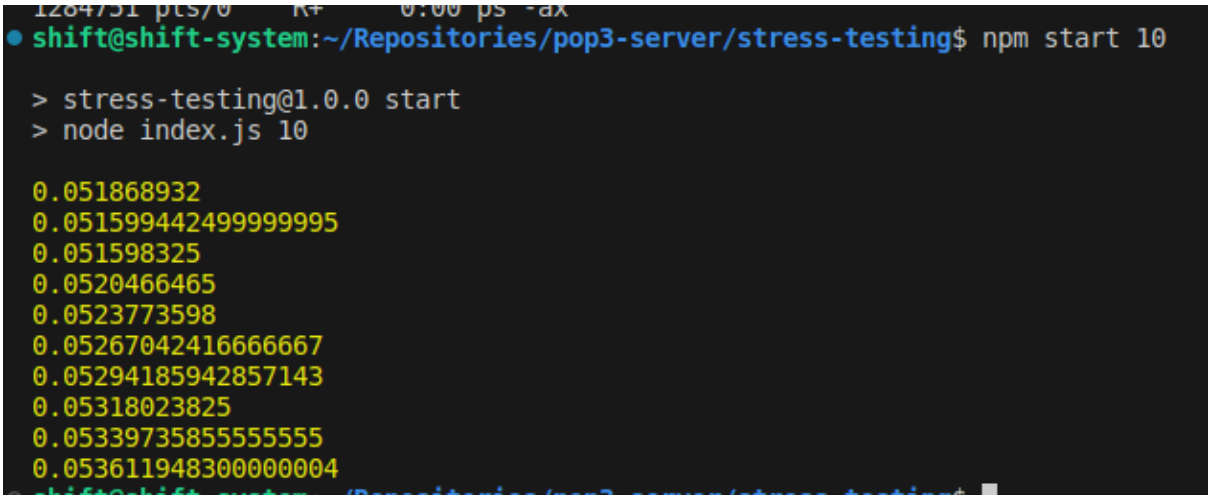




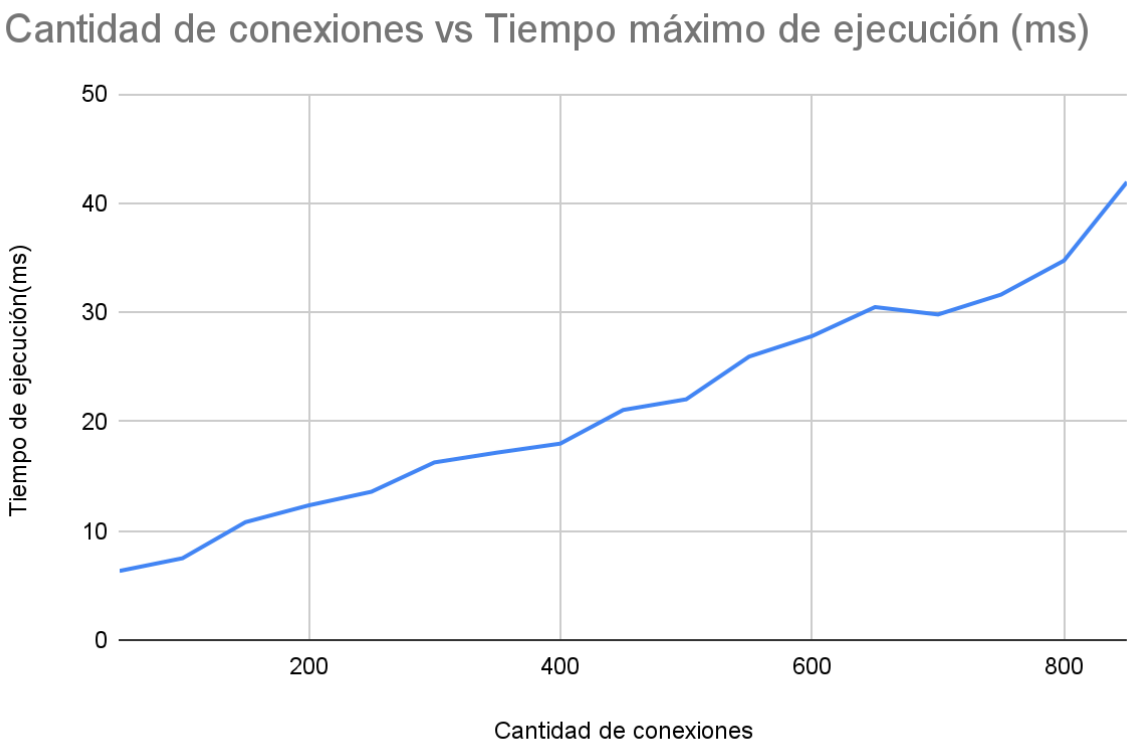
Cabe destacar que en todas las ejecuciones, el porcentaje de uso del CPU fue del 100% durante el procesamiento de las solicitudes.

7.4 Ejecuciones de gran cantidad de RETRs

Para realizar un buen testeo del throughput a medida que se suman conexiones, creamos un script JS encargado de realizar conexiones en simultáneo. Si se quieren testear N conexiones, se requieren de N usuarios existentes. Por eso se crearon dos scripts pequeños de bash encargados de rellenar los Mailbox con usuarios y archivos. El archivo Javascript después se encarga de agregar los users al server mediante PEEP y también realiza los llamados y su respectivo timing.
Por debajo podemos ver stress testing con 10 conexiones simultáneas.



Utilizando el valor de mayor tiempo de ejecución, podemos ver el gráfico de cómo se degrada el throughput en intervalos de 50 clientes.



8. Guía de instalación

1) Realizar `make all CC=gcc` o `make all CC=clang` para compilar tanto el servidor POP3 y PEEP como el cliente PEEP.
2) Ejecutar en una terminal, en el directorio raíz: `./pop3d -u user:password -u user2:password2 -m ../maildirectory`, esto comenzará el servidor POP3 y creará dos usuarios, user y user2. Además, establece el directorio de donde se obtienen los mails de los distintos usuarios como ../maildirectory. Más usuarios pueden ser añadidos a través de una conexión PEEP (utilizando el cliente PEEP).
3) Se puede utilizar cualquier MUA para acceder al servidor. Una forma rápida de acceder desde la terminal es mediante curl o directamente con netcat como `nc -C localhost 1110` (puerto default).
4) Para acceder a las opciones de administrador (PEEP), leer la siguiente parte de este documento. Si se quiere establecer las credenciales del administrador, utilizar el argumento `--peep-admin <admin-username>:<admin-password>`

9. Instrucciones de configuración

Si quisiera acceder al cliente de moderación y estadísticas, se debe ejecutar en otra terminal (o enviando el proceso pop3d a background) un comando del estilo,

`./peepclient -address localhost -port 2110 -user <admin-username> -pass <admin-password>`

El usuario y la contraseña default del admin son `root` y `root`… respectivamente.

Se pueden modificar estos valores default como fue indicado en el punto 4 de la Guía de Instalación.

10. Ejemplos de configuración y monitoreo

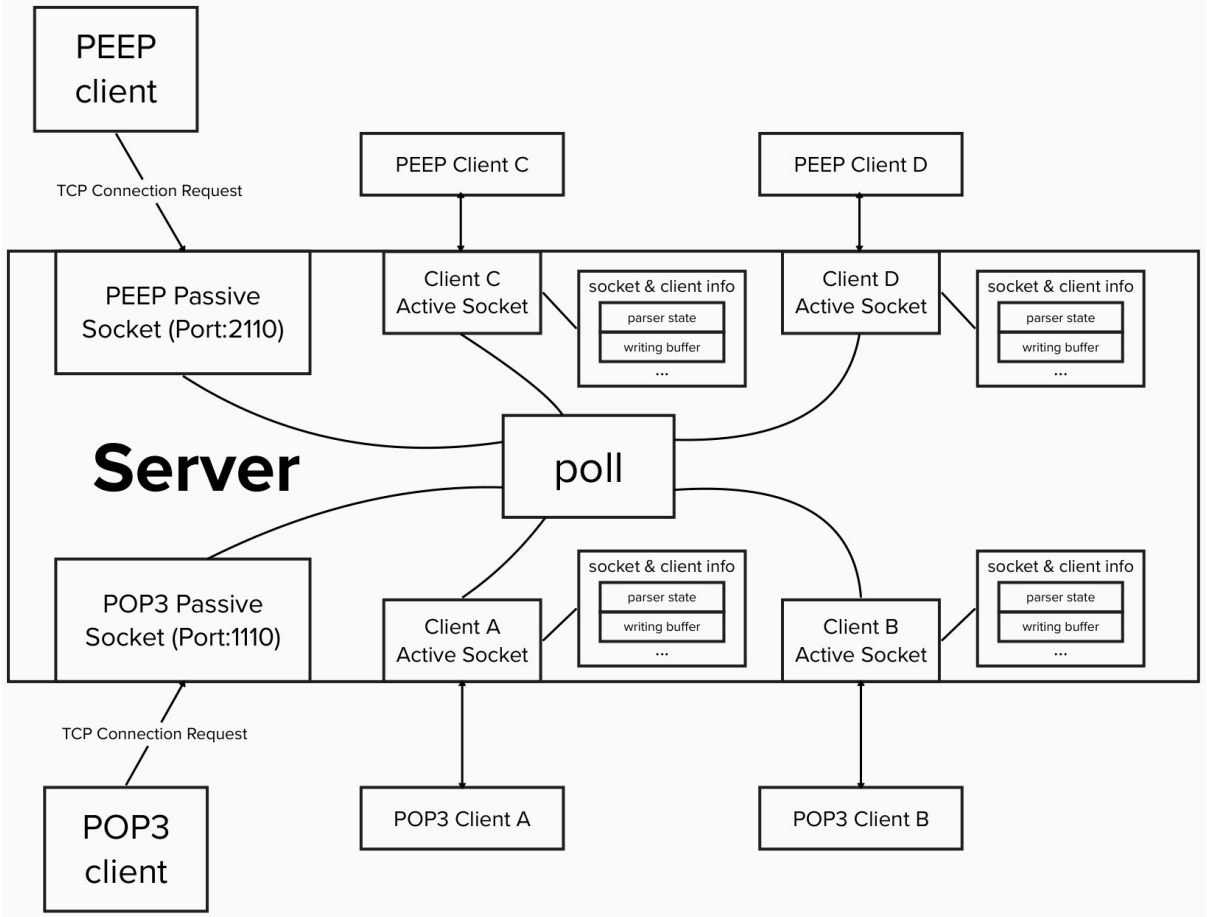
A continuación se mostrarán ejemplos de cómo se puede configurar y monitorear determinados parámetros del servidor POP3 de manera simple usando el cliente, sin la necesidad de tener que conocer el RFC que diseñamos. Los comandos los cuales fueron enviados por el usuario, son marcados con un >

```
mabaez@DESKTOP-M4AGRLN:~/protos/pop3-server$ ./peepclient -address localhost -port 2110 -user root -pass root
Connected successfully to server through PEEP!
See available commands with 'help'
>help
capa | quit | maildir | timeout | users | help | retrieved bytes | retrieved emails | max connections | set maildir <path> | set timeout <value> | delete user <username> | current connections amount | current logged users | removed emails amount | add user <username> <password> | set max connections <value> | all time connection amount | all time logged amount
>retrieved bytes
OK! 395
>timeout
OK! 0
>set timeout 10
OK!
>timeout
OK! 10
>maildir
OK! ../directories/



>current connections amount
OK! 0
>current logged users
OK!

>add user one one
OK!

>set timeout 0
OK!
>current connections amount
```

```
OK! 1
>current logged users
OK!  lines: 1
one

>set max connections 15
OK!
>delete user two
ERROR! Invalid username, does not exist
>add user hola chau
OK!
>delete user hola
OK!
>set maildir ~
ERROR! Invalid path format
>set maildir ../
OK!
>set maildir ../directories
OK!
>quit
OK!
```

11. Documento de diseño de proyecto

El proyecto puede subdividirse en 3 partes, primero el servidor POP3 el cual fue concebido siguiendo el RFC 1939. La parte de PEEP, el cual está íntimamente relacionada con POP3, ya que diseñamos el RFC de PEEP para el monitoreo y administración del mismo. Por último, tenemos el cliente PEEP, el cual actúa como intermediario entre el usuario y la parte PEEP del servidor.


PEEP client
TCP Connection Request
PEEP Client C
PEEP Client D
PEEP Passive Socket (Port:2110)
Client C Active Socket
socket & client info
parser state
writing buffer
...
Client D Active Socket
socket & client info
parser state
writing buffer
...
Server
poll
POP3 Passive Socket (Port:1110)
Client A Active Socket
socket & client info
parser state
writing buffer
...
Client B Active Socket
socket & client info
parser state
writing buffer
...
TCP Connection Request
POP3 client
POP3 Client A
POP3 Client B