Instituto Tecnológico de Buenos Aires

72.07 - Protocolos de Comunicación

Trabajo Práctico Especial

Autores

LIU, Jonathan Daniel - 62533

MARENGO, Tomas Santiago - 61587

VILAMOWSKI, Abril - 62495

WISCHÑEVSKY, David - 62494

Docentes

GARBEROGLIO, Marcelo (Teórica)

CODAGNONE, Juan F. (Laboratorio)

KULESZ, Sebastián (Laboratorio)

MIZRAHI, Thomas (Ayudante Alumno)

Noviembre 2023

Índice

1. Introducción

El trabajo práctico especial de la materia Protocolos de Comunicación consistió en la implementación de un servidor POP3 respetando los RFC 1939 [1] y 2449 [2], el desarrollo de un protocolo de monitoreo y configuración de este servidor (llamado DAJT) y la puesta en funcionamiento de un cliente que utilice este protocolo. En el siguiente informe se detallará el proceso de creación del servidor y el protocolo.

En la sección 2, se realizará una descripción de los mismos.

En la sección 3, se explicarán algunos problemas encontrados en el desarrollo y en caso de que se pueda, cómo se solucionaron.

En la sección 4, se plantearán algunas limitaciones del proyecto, principalmente relacionadas con detalles de implementación o capacidades del sistema.

En la sección 5, se propondrán posibles extensiones al servidor, que por cuestiones de tiempo o relevancia, se decidieron dejar de lado en esta entrega.

En la sección 6, se hará una breve conclusión, y luego en la sección 7 se detallarán algunas de las situaciones de pruebas relevantes, inspiradas en lo hablado durante las clases prácticas.

En las secciones 8, 9 y 10, se detallarán los pasos para instalar tanto el servidor POP3, como el cliente de DAJT y se proveerán las instrucciones para su configuración. Estas se verán acompañadas de ejemplos para facilitar su uso y comprensión.

Por último, en la sección 11, se hará una explicación de las etapas de diseño del proyecto, comentando el pasaje de un sistema bloqueante a uno no bloqueante, y la incorporación de las transformaciones. También se detallará el contenido de algunos archivos y se comentarán las estructuras utilizadas que resultaron imprescindibles para el desarrollo del proyecto.

2. Descripción detallada de los protocolos y aplicaciones desarrolladas

2.1. POP3

El protocolo POP3 es un protocolo de aplicación, orientado a texto y a conexión, y es confiable. Está destinado a permitir que una estación de trabajo acceda dinámicamente a un buzón de correo en un servidor anfitrión de una manera útil. Generalmente, esto significa que el protocolo POP3 se utiliza para permitir que una estación de trabajo recupere el correo que el servidor está reteniendo.

A lo largo de la sesión que conecta cliente y servidor, se transiciona entre los estados AUTHENTICATION, TRANSACTION y UPDATE. Según el RFC 1939, para que una aplicación servidor pueda ser considerada un servidor POP3, esta debe soportar por lo menos los comandos LIST, RETR, STAT, DELE, RSET, NOOP y QUIT. También debe tener al menos un mecanismo de autenticación. En este caso, los elegidos para el ingreso de credenciales fueron los comandos USER y PASS. El funcionamiento de todos estos comandos son detallados en el RFC 1939. El servidor implementado responde a los comandos anteriores, sumando el comando CAPA y la capacidad PIPELINING definidos en el RFC 2449.

Cuando el cliente se conecta al servidor, entra en el estado de AUTHENTICATION. Debe ingresar sus credenciales para poder acceder a su casilla. Para hacerlo, debe autenticarse ingresando el nombre de usuario y contraseña de la casilla de correo. Luego de ser autorizado, pasa al estado TRANSACTION. En este, el cliente podrá obtener información de su casilla de correo: listar (LIST), obtener (RETR) y eliminar mails (DELE + QUIT). Cuando el cliente cierre la sesión con el comando QUIT, se eliminarán los mails que decidió borrar utilizando el comando DELE. En caso de cerrar la sesión antes de autenticarse, se cerrará la conexión POP3 de la misma forma.

El servidor POP3 de este trabajo práctico está basado en una implementación no bloqueante, gracias a la utilización de la función select [3] y la API de selector.

Por otro lado, el servidor soporta el uso de aplicaciones tranformadoras de mails, que permiten que los usuarios reciban una versión modificada de los mismos cuando utilicen el comando RETR.

El servidor cuenta con un sistema de logging por niveles, no bloqueante y no persistente, y exige indicar por argumentos los usuarios de la casilla de correo con sus correspondientes contraseñas y la ubicación del directorio Maildir.

2.2. DAJT

Además de la implementación del servidor previamente descrito, se implementó un servicio de monitoreo y configuración, que permite averiguar estadísticas y cambiar propiedades del servidor POP3. Este servicio respeta el protocolo DAJT, definido en el RFC adjunto en la entrega. Este protocolo es orientado a texto y conexión, basado sobre TCP.

Para facilitar el uso del protocolo, se desarrolló una aplicación cliente (KERL) que recibe los comandos como paths y argumentos al estilo CURL [4], se encarga de armar el formato de cada petición según el RFC y, al recibir una respuesta del servidor, imprime el resultado por salida estándar.

En esta versión, el servidor responde a las siguiente peticiones:

1. Autenticar a un usuario.
2. Modificar el tamaño del buffer.
3. Consultar el tamaño del buffer.
4. Consultar el total de bytes enviados en el servidor POP3.

5. Consultar la cantidad de conexiones actuales en el servidor POP3.
6. Consultar la cantidad de conexiones totales en el servidor POP3.
7. Habilitar y deshabilitar las transformaciones de correo.
8. Consultar el estado de las transformaciones.
9. Modificar el transformador de correo.
10. Cerrar la sesión con el servidor DAJT.

Otra de las capacidades con las que cuenta el servidor es el PIPELINING, que agiliza la comunicación entre ambos hosts. Uno de los comandos de los que se puede sacar provecho gracias al PIPELINING es STAT, que devuelve estadísticas puntuales dependiendo del parámetro que se le envíe.

3. Problemas encontrados durante el diseño y la implementación

Para implementar el servidor de manera no bloqueante, se decidió utilizar el selector provisto por la cátedra junto con la implementación de stm. Por lo que el primer desafío fue comprender totalmente la implementación para sacarle el máximo partido y para poder realizarles cambios sin generar mayores problemas.

Otro de los desafíos de diseño que conllevó el pasaje de un sistema bloqueante a uno no bloqueante fue cómo plantear el flujo de la información, ya que esto podía implicar mantener un estado o progreso parcial de este. Además, se tomó un enfoque *greedy* para los flujos, permitiendo que cada vez que el selector elige un file descriptor, no se discrimina cuál fue, y se intentan realizar todos los flujos posibles si las condiciones eran suficientes. Esto implica que las operaciones de lectura y escritura puedan arrojar el error EWOULDBLOCK o EAGAIN, y se deban manejar correctamente.

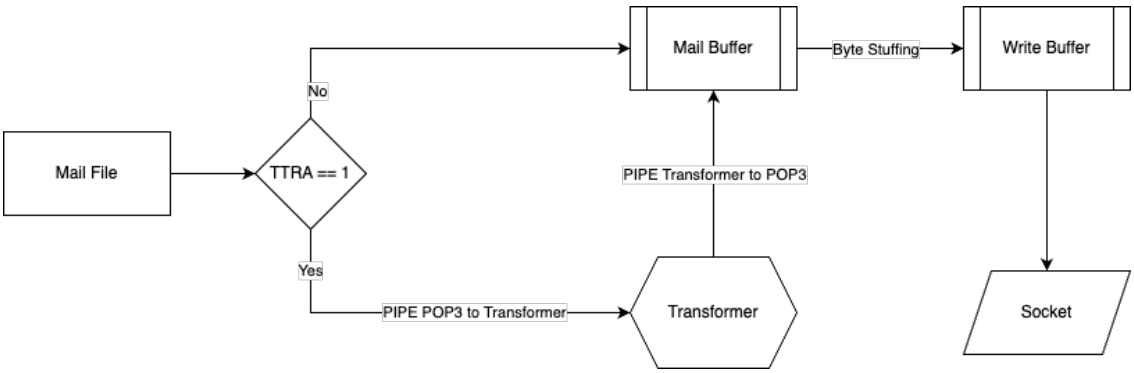

Figura 1: Diagrama de flujo de las transformaciones.

Debido a que se decidió no utilizar un buffer persistente entre la lectura de archivo del mail y el pipe de entrada del transformador, se tuvo que recurrir a la función *lseek* [5], para deshacer la lectura del buffer de mail, seguido de desuscribir de lectura al archivo del mail, hasta que se pueda escribir en el pipe.

Otro de los problemas que se hizo presente en el desarrollo del trabajo práctico, fue que resultó imposible desarrollar el proyecto en MacOS, ya que el sistema operativo no reconoce algunas librerías, y por lo tanto tampoco sus funciones y constantes, lo que imposibilita la compilación de la aplicación. Se intentó utilizar una máquina virtual en su lugar pero la lentitud de esta era notable. Este inconveniente fue comentado a los profesores en clase.

4. Limitaciones de la aplicación

Tanto el servidor implementado como el protocolo de comunicación presentan una serie de limitaciones que se detallarán a continuación.

Hay una limitación global que proviene de *FD_SETSIZE* de select.h. Esta variable capea la cantidad de file descriptors (en 1024) con los que se puede inicializar nuestro selector, por lo que el total de file descriptors entre los dos protocolos es 1024. Sabiendo que, por ejemplo, un usuario que está leyendo un mail ocupa 2 file descriptors, la cantidad de usuarios máxima en este escenario sería de 512.

4.1. POP3

Una limitación tiene que ver con la cantidad de usuarios concurrentes. Para poder hacer un "garbage collector" de usuarios, se creó un vector que soporta 512 clientes (POP3 + DAJT). En caso de que se quiera escalar mejor, podría ser recomendable incorporar una lista doblemente enlazada o algún tipo de estructura más compleja que permita llevar registro de los mismos.

Otra limitación es la cantidad de caracteres que se pueden enviar al realizar un comando. Tanto el parser de POP3 como DAJT manejan en total 255 caracteres como máximo, por lo que si se intenta enviar un comando de una extensión mayor se arrojará un error. Lo mismo aplica para comandos de menos de 4 caracteres. Si bien es una limitación, ningún comando más argumentos en un escenario que se ajuste al RFC 1939 y 2449 llegaría a tal longitud.

La limitación más importante del proyecto, tiene que ver con que se decidió no hacer una distinción entre archivos que terminan con o sin CRLF. Esto provoca que, a diferencia de dovecot, se agregue en todos los casos un '\r\n .\r\n' al final de la respuesta del RETR. Se decidió hacer esto por dos motivos: en primer lugar, porque garantiza que el cliente pueda reconocer en todos los casos de manera correcta el fin del mensaje; y en segundo, porque implentarlo requeriría incorporar una validación ad hoc muy compleja, que perjudicaría la performance de la implementación actual.

4.2. DAJT

Una limitación tiene que ver con el tamaño de los buffers: dado que la variable de tamaño de buffer se comparte tanto para DAJT como POP3, se tuvo que poder garantizar que los administradores no puedan cambiarla a una cantidad de caracteres menor a los mínimos mensajes de ambas. De esta manera, se decidió utilizar un mínimo de 128 bytes, que cubre ambos casos satisfactoriamente.

Otra limitación está relacionada con la libertad que tienen los usuarios para asignar transformadores en el comando TRAN: no existe forma de validarlo correctamente antes de asignarlo (no alcanza con verificar que el archivo exista o sea ejecutable pues puede ser modificado o borrado antes de ejecutar RETR). Y aún más, una vez que se hizo fork y exec, es muy difícil verificar si bash logró ejecutar el comando correctamente. Se decidió ir por la solución más sencilla, y si el transformador falla por cualquier motivo, simplemente se considerará que la comunicación terminó por EOF y por lo tanto la transformación aplicada será equivalente a una que no escriba nada al pipe, es decir, que devuelva un mensaje vacío.

5. Posibles extensiones

5.1. POP3

Actualmente, salvo por el comando CAPA y el uso de USER y PASS, la implementación del servidor POP3 cumple sólo con lo básico según el RFC 1939. En una implementación posterior podrían incorporarse el resto de los comandos presentes en los RFCs 1939 y 2449, tales como TOP, UIDL o APOP. También podría soportar SSL/TLS para garantizar la seguridad de los datos.

Otra posible incorporación para una extensión del protocolo, es el cierre de sesión automático por inactividad. Como menciona el RFC 1939, si se quiere implementar, se debe esperar que pase un tiempo determinado (al menos diez minutos). Si se recibe cualquier comando en el medio, se debe reiniciar el temporizador. Una vez que el tiempo se haya acabado, se debe cerrar la conexión TCP, sin eliminar mensajes ni enviar una respuesta al cliente.

Por otra parte, se podría reincorporar la implementación de *advisory locks* en la carpeta de cada usuario cuando estos inician sesión (que se había hecho durante la primera parte del proyecto), para garantizar la consistencia de los archivos al tener muchos usuarios concurrentes.

Una última cuestión a mejorar, es que a diferencia de Dovecot, el servidor no realiza movimiento de archivos entre las carpetas de /cur, y /new. Se decidió no incorporar esta funcionalidad pues no se consideró relevante y no se considera parte de POP3.

5.2. DAJT

El comando STAT puede ser ampliado para agregar una mayor variedad de estadísticas. Ahora mismo, únicamente se usan 3 posibles caracteres de entrada del comando, de manera que siempre y cuando se le asigne un carácter distinto a la nueva estadística, esta debería poder ser procesada.

Por otra parte, en un principio se había pensado implementar un comando para manejar timeouts del cliente, pero se descartó por cuestiones de tiempo. Este comando tendría un funcionamiento similar a TRAN o BUFF en el sentido que, en caso de no tener parámetros podría mostrar el tiempo actualmente configurado, y con parámetros permitiría cambiar el valor dentro del dominio.

Otro aspecto deliberadamente omitido luego de ser hablado en las clases fue el manejo de usuarios de POP3 dentro de DAJT: borrado, agregado, cambio de contraseña y listado. Esta funcionalidad fue descartada ya que agregaría la necesidad de manejar respuestas multilínea, además que sería altamente propensa a condiciones de carrera.

6. Conclusiones

Gracias a este trabajo práctico, se logró afianzar los conocimientos adquiridos a lo largo del cuatrimestre. Entre ellos, se encuentran la comunicación entre clientes y servidores, el funcionamiento de protocolos de aplicación y transporte, la comprensión e interpretación de documentos RFC, la programación con sockets pasivos y activos y sobre procesos bloqueantes, no bloqueantes y selectores.

A su vez, el desarrollo del trabajo práctico demostró lo importante que es seguir estándares para que los protocolos sean retrocompatibles, y tengan en cuenta futuras extensiones.

En las secciones siguientes, se analizarán cuestiones de implementación, ejemplos y cómo útilizar los servidores para cada protocolo y el cliente DAJT.

7. Ejemplos de prueba

7.1. Usando POP3 con Thunderbird

Se probó el servidor desarrollado con el MUA Thunderbird. Para esto, se agregó una cuenta con la siguiente configuración:

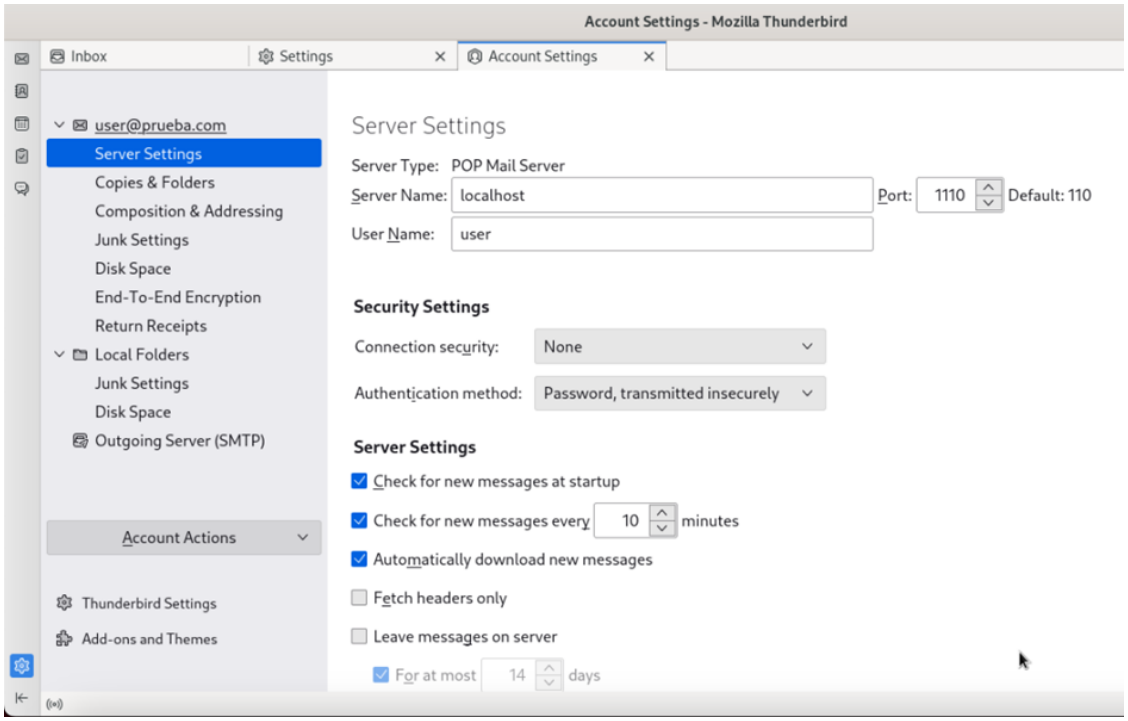

Figura 2: Mozilla Thunderbird UI

Se observa también que en principio hay 5 mails dentro de la carpeta *user*.

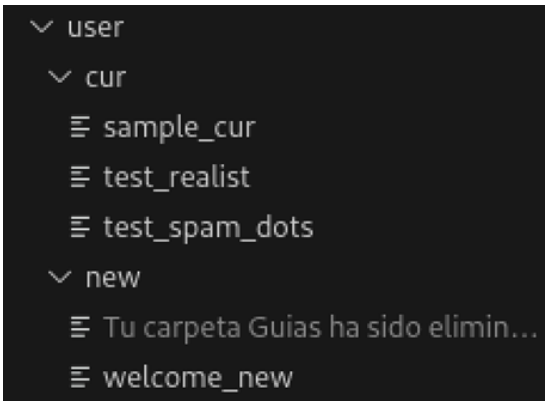

Figura 3: Emails del usuario en formato maildir

Una vez que se inicia el servidor con el comando

```
./pop3d -p 1110 -P 9090 -u user:pass -d .
```

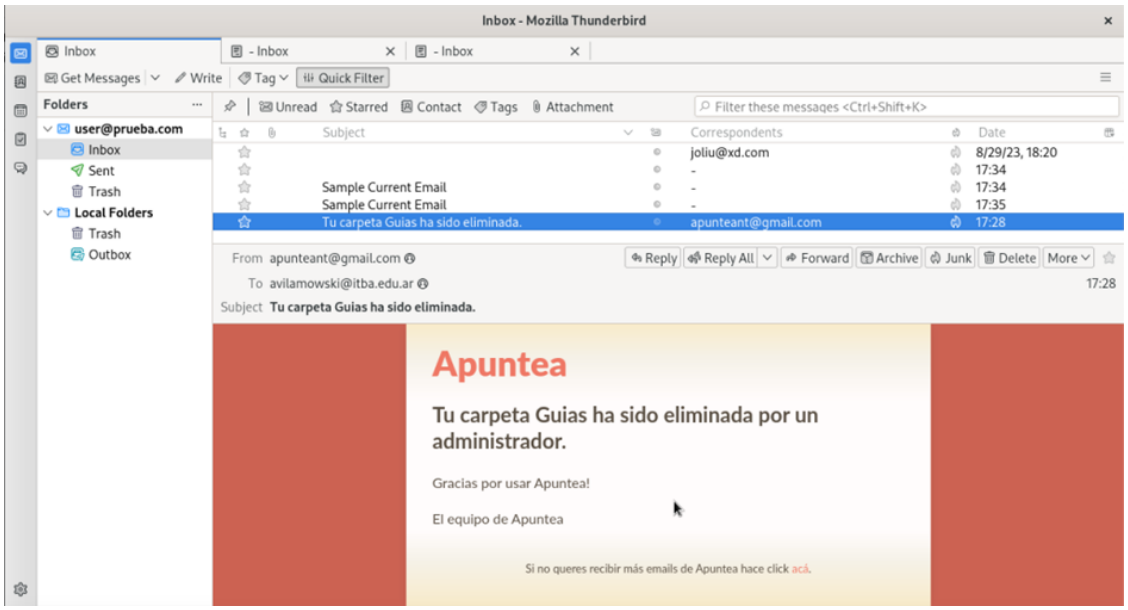

Figura 4: Emails del usuario en Thunderbird

7.2. Tamaño del Buffer con un usuario

Para testear el tamaño de los buffers, se utilizó un archivo de 1GB generado con el siguiente comando:

```
dd if=/dev/urandom of=file bs=4096 count=250000
```

Luego, se fue cambiando el tamaño de los buffers entre los siguientes almacenamientos:

```
~ time curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  980M    0  980M    0     0  72.2M      0 --:--:--  0:00:13 --:--:-- 72.9M
curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null  2.01s user 1.04s system 22% cpu 13.580 total
```

Figura 5: Buffer 1024 bytes

```
~ time curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  980M    0  980M    0     0   129M      0 --:--:--  0:00:07 --:--:--  132M
curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null  1.84s user 0.70s system 33% cpu 7.567 total
```

Figura 6: Buffer 2048 bytes

```
~ time curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  980M    0  980M    0     0   230M      0 --:--:--  0:00:04 --:--:--  230M
curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null  1.75s user 0.44s system 51% cpu 4.262 total
```

Figura 7: Buffer 4096 bytes

```
[➜  ~ time curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  980M    0  980M    0     0   346M      0 --:--:--  0:00:02 --:--:--  346M
curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null  1.72s user 0.30s system 70% cpu 2.843 total
```

Figura 8: Buffer 8192 bytes

```
[➜  ~ titime curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  980M    0  980M    0     0   464M      0 --:--:--  0:00:02 --:--:--  466M
curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null  1.71s user 0.17s system 88% cpu 2.126 total
```

Figura 9: Buffer 16384 bytes

```
[➜  ~ time curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  980M    0  980M    0     0   471M      0 --:--:--  0:00:02 --:--:--  472M
curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null  1.70s user 0.16s system 89% cpu 2.095 total
```

Figura 10: Buffer 32768 bytes

```
[➜  ~ time curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  980M    0  980M    0     0   454M      0 --:--:--  0:00:02 --:--:--  455M
curl pop3://user:pass@192.168.0.177:1110/4 > /dev/null  1.71s user 0.22s system 88% cpu 2.167 total
```

Figura 11: Buffer 65536 bytes

Resumiendo:

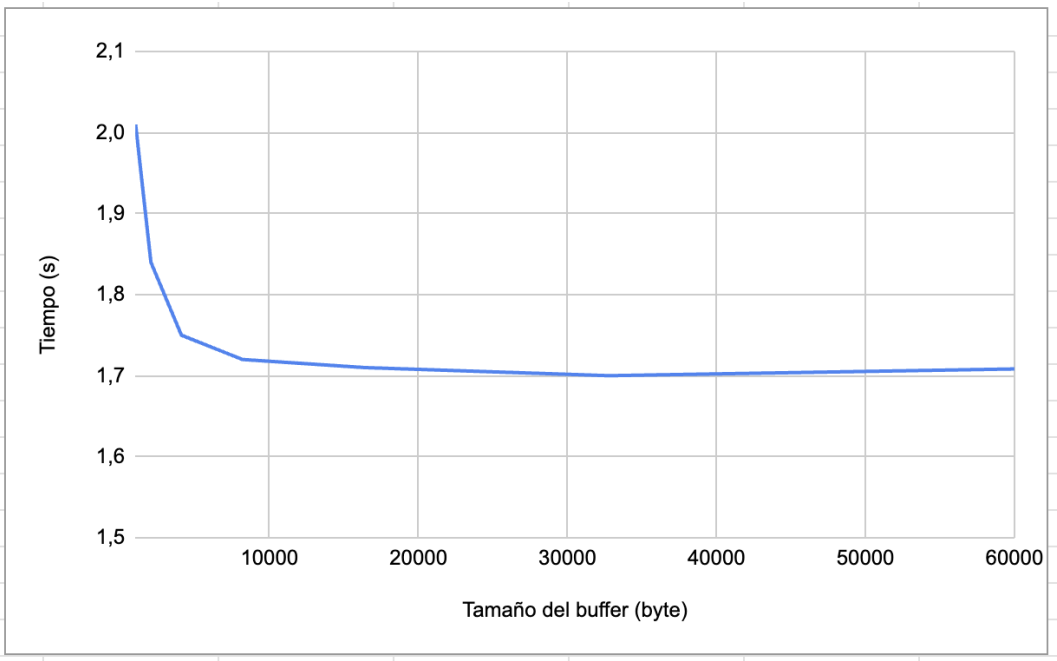

Figura 12: Tamaño del buffer vs Tiempo

Se concluyó que a partir de los 8192 bytes, el tiempo que tardó en realizar el procesamiento se mantuvo casi constante, por lo que ese tamaño de buffer fue el que se eligió por defecto.

7.3. Comparación con Dovecot

Para comparar con Dovecot, se utilizó un archivo generado con los siguientes comandos:

```
dd if=/dev/urandom of=file bs=4096 count=250000
base64 file > file64
```

Se generó el archivo en formato base 64, pues su contenido tiene caracteres legibles. El archivo resultante tiene un tamaño de aproximadamente 1.35 GB.

Se compararon los tiempos de dovecot contra los de nuestro servidor con tamaño de buffer 8192, en una misma computadora la acción de RETR para el archivo en formato base 64, ejecutando:

```
#servidor dovecot
time curl pop3://user:pass@localhost:110/1 > /dev/null
#servidor nuestro
time curl pop3://user:pass@localhost:1110/1 > /dev/null
```

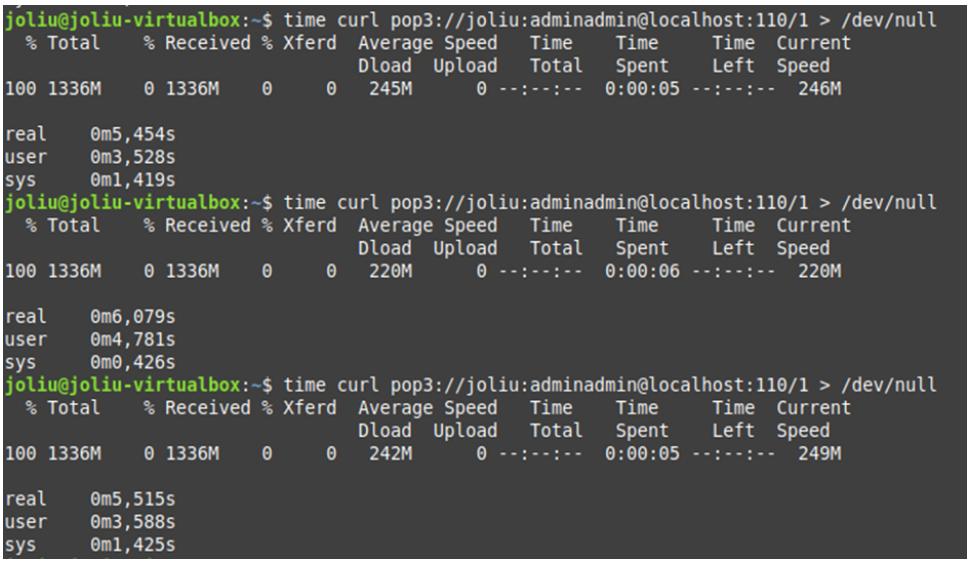

Figura 13: Tiempos de Dovecot

```
joliu@joliu-virtualbox:~$ time curl pop3://user:1234@localhost:1110/1 > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1336M    0 1336M    0     0   259M      0 --:--:--  0:00:05 --:--:--  261M

real    0m5,157s
user    0m4,272s
sys     0m0,686s
joliu@joliu-virtualbox:~$ time curl pop3://user:1234@localhost:1110/1 > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1336M    0 1336M    0     0   212M      0 --:--:--  0:00:06 --:--:--  213M

real    0m6,290s
user    0m5,158s
sys     0m0,505s
joliu@joliu-virtualbox:~$ time curl pop3://user:1234@localhost:1110/1 > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1336M    0 1336M    0     0   234M      0 --:--:--  0:00:05 --:--:--  239M

real    0m5,710s
user    0m4,652s
sys     0m0,555s
joliu@joliu-virtualbox:~$ time curl pop3://user:1234@localhost:1110/1 > /dev/null
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1336M    0 1336M    0     0   244M      0 --:--:--  0:00:05 --:--:--  247M

real    0m5,483s
user    0m4,304s
sys     0m0,831s
```

Figura 14: Tiempos de nuestro servidor

Se obtuvieron resultados muy similares (una diferencia menor de 0.5s, alrededor del 10 % del tiempo de Dovecot), considerando que se hicieron varias mediciones y los valores obtenidos variaban dentro de un rango de 1s.

7.4. Pipelining

Se probó el correcto funcionamiento del pipelining en pampero de la siguiente manera.

```
[dwischnevsky@pampero ~]$ printf "CAPA\nUSER user\npass pass\nLIST\nQUIT\n" | ncat localhost 1110 -C
+OK POP3 Party Started
+OK
CAPA
USER
PIPELINING
.
+OK
+OK Logged in.
+OK 7 messages:
1 201326592
2 218103808
3 370
4 1024000000
5 73
6 29
7 0
.
+OK POP3 Party over (7 messages left)
[dwischnevsky@pampero ~]$
```

Figura 15: Funcionamiento de Pipelining

El comportamiento fue el esperado.

7.5. Lecturas parciales (stty)

El correcto funcionamiento de las lecturas parciales se probó de la siguiente manera:

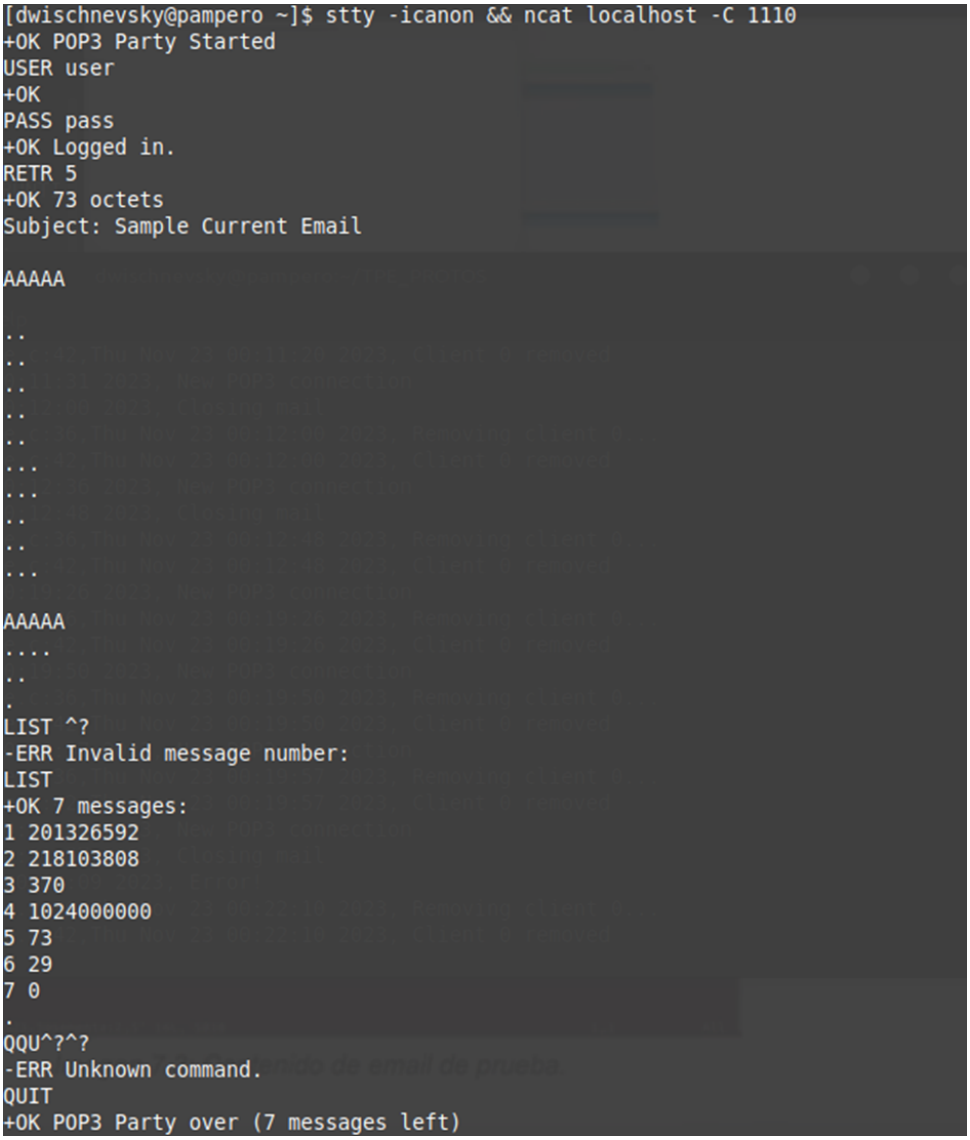

Figura 16: Funcionamiento de lecturas parciales

Como se puede ver, el comportamiento es el esperado.

7.6. Escrituras parciales

Para comprobar el correcto funcionamiento de las escrituras parciales, se modificaron todas las syscalls de send para que como máximo escriban un byte. Se encontraron algunos errores con la finalización de los comandos, que fueron solucionados exitosamente.

7.7. Integridad de los mails

Para verificar la integridad de la información en los mails, se realizaron dos pruebas.

1. Se creo el archivo *test_spam_dots*, que cuenta con varias situaciones propensas a fallar por byte stuffing.

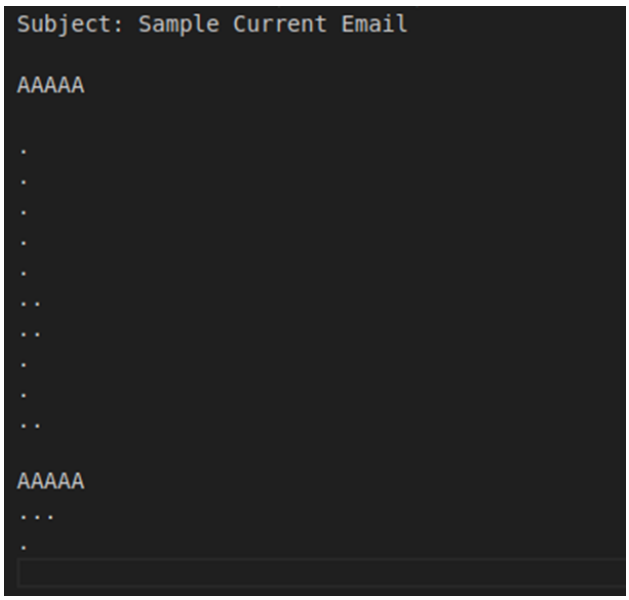

Figura 17: Email con muchos puntos

Se ejecutó RETR y se lo comparó con el archivo:

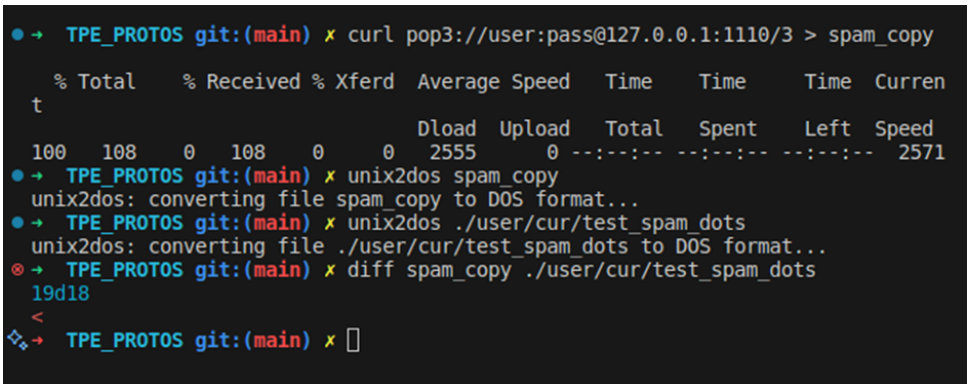

Figura 18: Resultado test a

El diff muestra que existe un CRLF de más, cosa esperable y ya discutida.

2. Se utilizó uno de los archivos de 1GB de los tests anteriores:

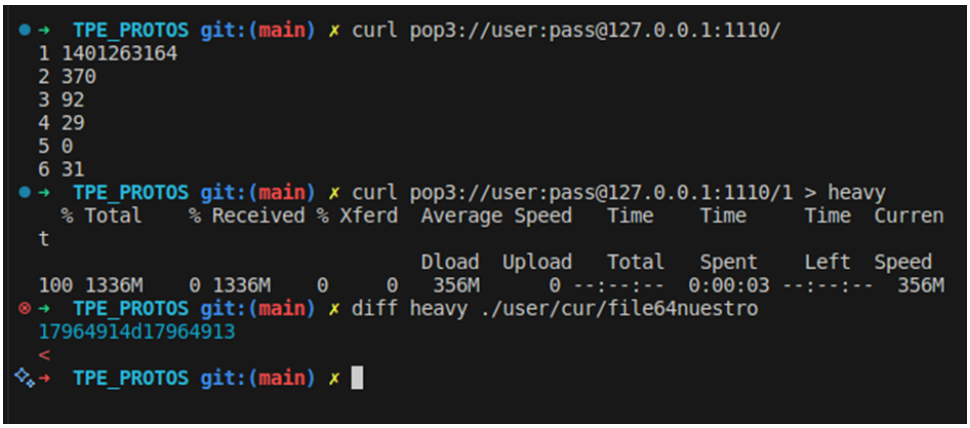

Figura 19: Resultado test b

7.8. Interrupción de lectura de clientes

Una de las pruebas realizadas consistió en conectar a dos clientes descargando mails de 1GB.

La intención fue probar que la memoria fuera liberada y el servidor continuara sin problemas cuando:

a) Se interrumpe el servidor con CTRL + C

b) Se interrumpe a los clientes con CTRL + C

El desarrollo ocurrió de la manera esperada en ambos casos.

7.9. 500 conexiones

Una de las pruebas de stress se realizaron con la librería *node-pop3* a partir del script stress.js (que se puede encontrar en el Anexo). Estas consistieron, en primer lugar en ejecutar user y pass en 500, 501 y 600 conexiones sobre el mismo usuario (en el último caso se ignoraban a los últimos 88 clientes). Luego, se agregó un RETR de un archivo de 370 bytes, a cada una , y se forzó a que los clientes mantuvieran abierta la conexión usando comandos sleep.

El comportamiento fue el esperado: el servidor logró manejar todas las peticiones y luego de terminar, cuando se mató al proceso, se pudo ver que no se presentaron memory leaks.

Por último, se probó con archivos de 1MB, 10MB y 100MB. Los primeros dos funcionaron de la manera esperada, pero el último solo funcionó con hasta 200 usuarios. Con 500 usuarios, el sistema operativo cortó la ejecución del proceso, probablemente porque consideró que no respondía.

7.10. Tests con JMeter

Para poder testear con JMeter usuarios concurrentes y poder obtener el throughput utilizamos el Plugin Manager de JMeter y con este agregamos el plugin Concurrency Thread Group [6].

Dentro de un test se agregó una instancia de este plugin, y se lo configuró de esta manera (variando el Target Concurrency, es decir, la cantidad de usuarios máxima):

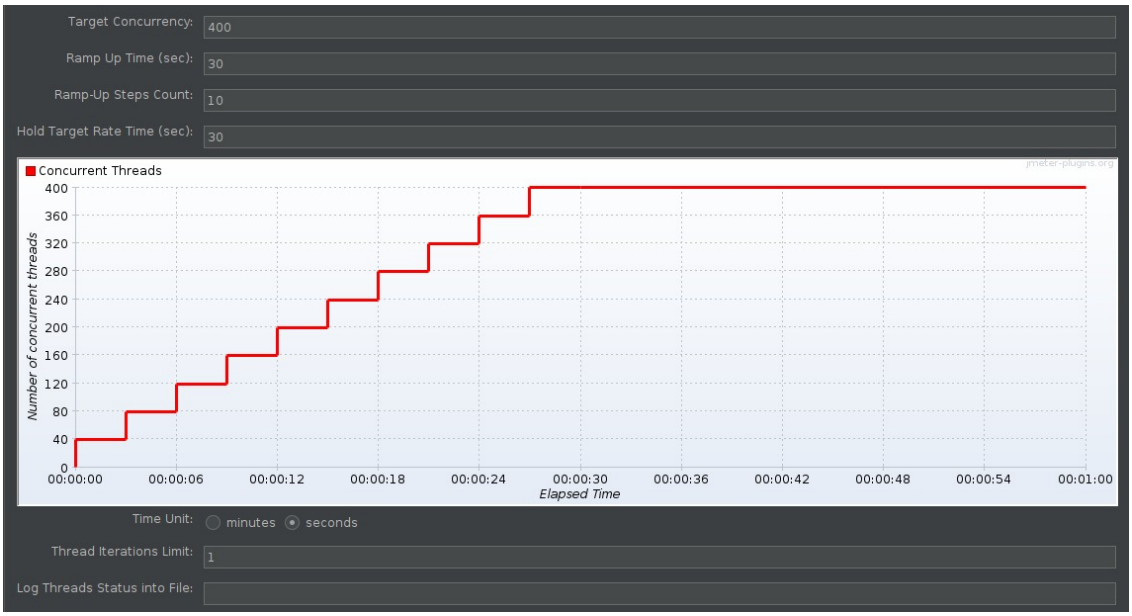

Figura 20: Configuración de Concurrency Thread Group

Luego, se configuró el Mail Reader Sampler de esta manera:

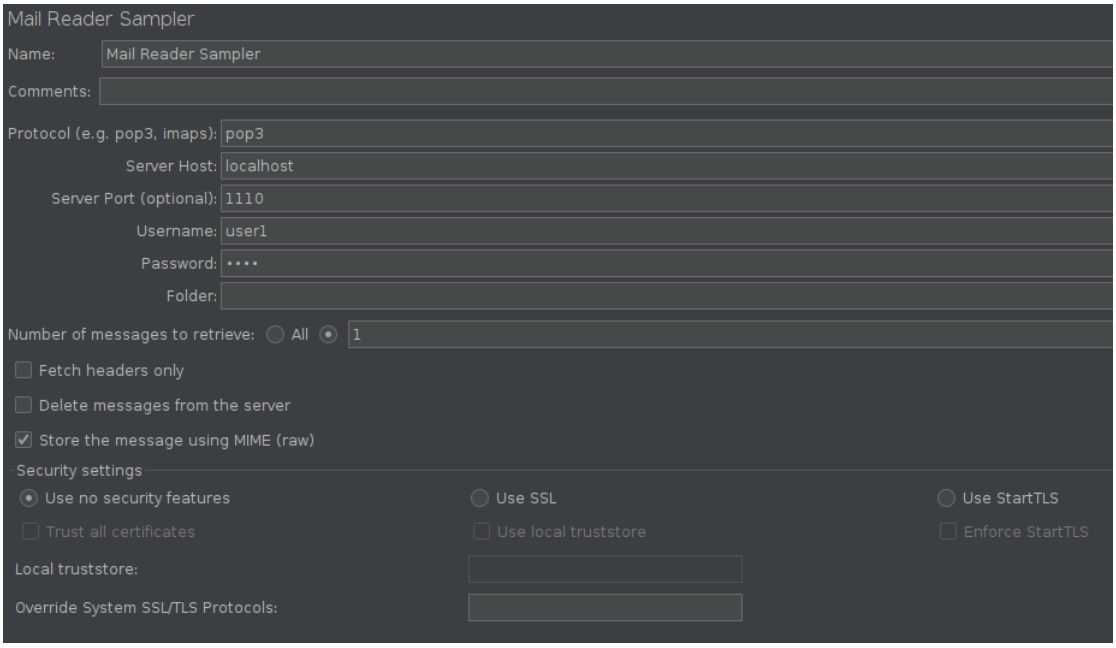

Figura 21: Configuración del Mail Reader Sampler

Por último, con el uso de distintos listener se obtuvieron los siguientes resultados para 400 usuarios leyendo cada uno un mail de 12MB.

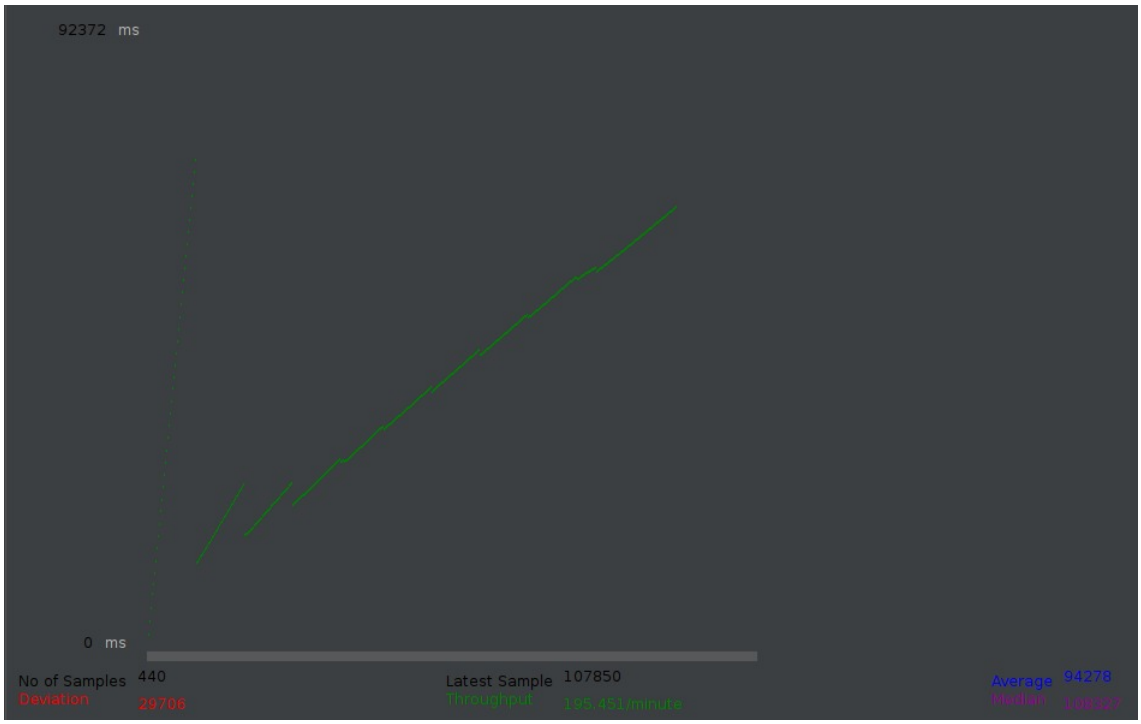

Figura 22: Troughput con buffer: 512 bytes

# Muestras	Promedio	Error	Throughput	KB/sec recibidos	Prom. Bytes
440	94278	0.00 %	3.3/seg	45399,86	14271418.0

Cuadro 1: Resultados con buffer: 512 bytes

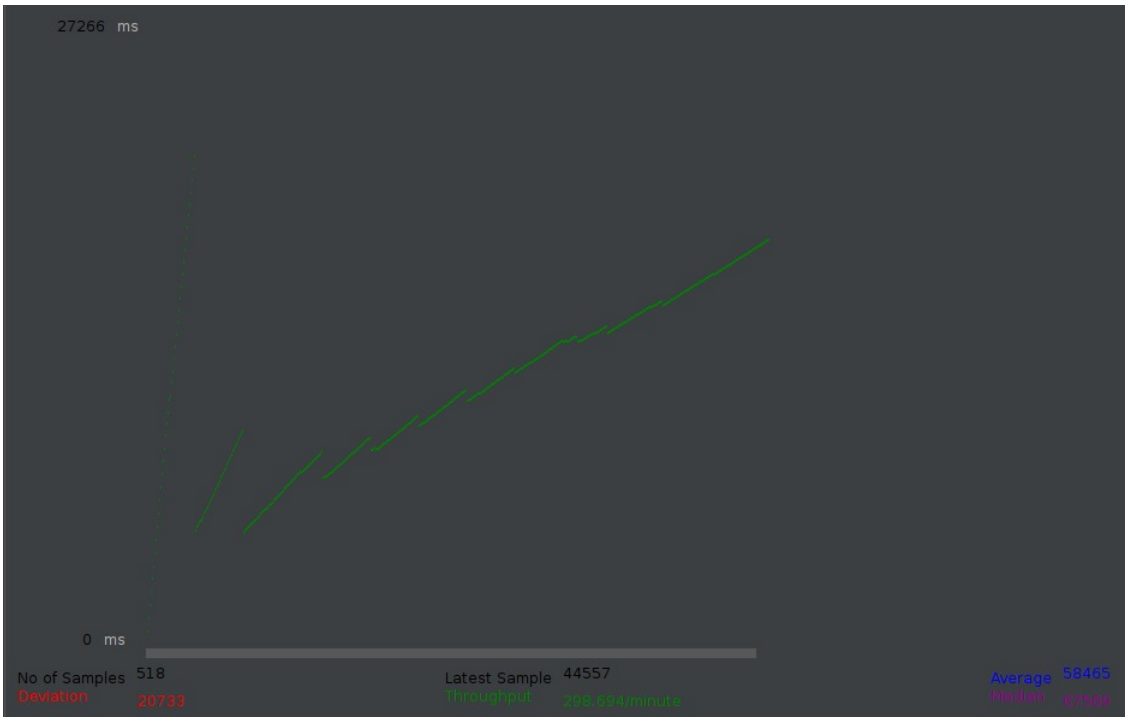

Figura 23: Troughput con buffer: 1024 bytes

# Muestras	Promedio	Error	Throughput	KB/sec recibidos	Prom. Bytes
518	58465	0.00 %	5.0/seg	69381.20	14271400.0

Cuadro 2: Resultados con buffer: 1024 bytes

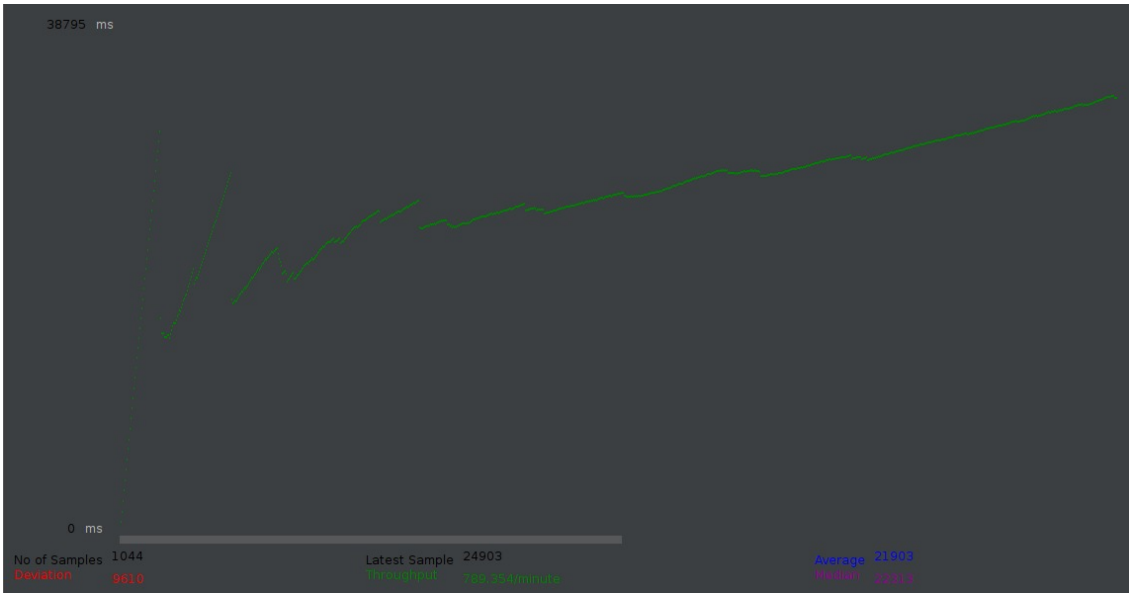

Figura 24: Troughput con buffer: 32768 bytes

# Muestras	Promedio	Error	Throughput	KB/sec recibidos	Prom. Bytes
1044	21903	0.00 %	13.2/seg	183353.04	14271425.0

Cuadro 3: Resultados con buffer: 32768 bytes

Luego, para 500 usuarios pero ahora leyendo cada uno un mail de 1,2MB (10 veces menos):

Figura 25: Troughput con buffer: 32768 bytes

# Muestras	Promedio	Error	Throughput	KB/sec recibidos	Prom. Bytes
9824	5066	0.00 %	147.9/seg	201051.62	1391995.0

Cuadro 4: Resultados con buffer: 32768 bytes

Se optó por testear 400 usuarios en el primer test por problemas de HEAP SPACE del JMeter. Cuando los mails pesaban 12MB se guardaban como MIME para comprobar que efectivamente se estaban mandando los bytes, por lo que se necesitó agrandar el HEAP hasta 16GB.

Resumiendo, lo que nos permite observar todo esto son varias cuestiones:

1. No se observan errores en un manejo elevado de usuarios concurrentes leyendo un mail de gran tamaño.
2. Para un mismo testeo, el throughput disminuye cuando disminuye el tamaño del buffer, ya que es más costoso el tener que ir más veces al select.
3. Para un mismo tamaño de buffer, el throughput aumenta si disminuye el tamaño promedio de los mails. Esto es así ya que termina más rápido de leer y mandar los mails y puede atender a más usuarios por segundo.
4. La cantidad de muestras (request de usuarios) es proporcional al throughput para una misma cantidad de tiempo de testeo (en nuestro caso es de 1 minuto). Esto es algo obvio pero es para entender la definición de muestras.

8. Guía de instalación

Dentro del mismo, la compilación del proyecto se realiza de la siguiente manera (cambiando compiler por GCC o Clang si se quiere usar otro que CC):

```
make all CC=<compiler>
```

Luego, se debe ejecutar el archivo generado, de nombre pop3d.

Sus flags son los siguientes:

- -d $< dir >$: ubicación del Maildir. Dentro de dicho directorio, se encuentran los directorios de los mails de los usuarios, cada uno con su /cur y /new, de donde se leerán los mails.
- -u $< user : pass >$ (opcional): permite agregar un usuario de pop3. Sus mails serán obtenidos de la carpeta con su mismo username. Por defecto pueden agregarse hasta 10 usuarios distintos.
- -U $< user : pass >$ (opcional): permite agregar un administrador de DAJT. No existen diferencias entre los administradores.
- -t $< transformador >$ (opcional): permite aplicar un transformador a los mensajes obtenidos en RETR.
- -p $< puerto >$ (opcional): permite setear el puerto del servidor de pop3. Por defecto es el 1110.
- -P $< puerto >$ (opcional): permite setear el puerto del servidor de DAJT. Por defecto es el 9090.
- -h: muestra una descripción de todos los comandos previamente mencionados y su funcionamiento.
- -v: muestra una breve descripción sobre el protocolo y su versión.

9. Instrucciones para la configuración

Para compilar el cliente, se debe acceder desde la terminal a la carpeta client dentro de src y compilar de la siguiente manera, modificando la palabra compiler por gcc o clang:

```
make all CC=<compiler>
```

Si se quisiera acceder al cliente para realizar modificaciones de configuración o mostrar estadísticas, se debe ejecutar en esa misma terminal el siguiente comando, rememorando al cliente CURL:

```
./kerl [-v4|-v6|-h] dajt://<usuario>:<contraseña>@127.0.0.1:9090/
```

Dependiendo del tipo de IP que se quiere usar, se puede poner el flag -v4 en caso IPv4 y -v6 en caso de IPv6. Por defecto, se utilizará IPv4. El flag -h muestra las rutas soportadas por el servidor.

Luego del último '/' se debe ingresar el comando a ejecutar. Las opciones posibles se enunciarán en la siguiente sección. Este path puede o no terminar en un y sólo un '/'.

10. Ejemplos de configuración y monitoreo

El cliente de configuración y monitoreo permite obtener estadísticas del servidor y realizar cambios en tiempo real. A continuación, se muestran algunas acciones que puede realizar el cliente:

Para mostrar el total de bytes enviados en el servidor POP3 debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/stat/b
```

Para mostrar la cantidad de conexiones actuales en el servidor POP3 se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/stat/c
```

Para mostrar la cantidad de conexiones totales en el servidor POP3 se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/stat/t
```

Para mostrar el almacenamiento actual del buffer se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/buff
```

Para actualizar el almacenamiento del buffer se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/buff/[number]
```

Siendo [number] reemplazado por un número que represente el nuevo almacenamiento en bytes que se desea para el buffer.

Ejemplo:

```
./kerl -v4 dajt://admin:admin@127.0.0.1:9090/buff/1234
Server Response: >OKE
```

Para desactivar las transformaciones se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/ttra/0
```

Para activar las transformaciones se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/ttra/1
```

Para ver el estado de las transformaciones se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/ttra
```

Para ver el transformador actual se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/tran
```

Para establecer un nuevo transformador se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/tran -body "commands"
```

Siendo la palabra commands reemplazada por el transformador y sus correspondientes argumentos

Ejemplos:

```
./kerl -v4 dajt://admin:admin@127.0.0.1:9090/tran -body "cat"
Body: cat
Server Response: >OKE
```

```
./kerl -v4 dajt://admin:admin@127.0.0.1:9090/tran -body "tr a-z A-z"
Body: tr a-z A-z
Server Response: >OKE
```

Para cerrar la sesión se debe correr:

```
./kerl dajt://admin:admin@127.0.0.1:9090/quit
```

11. Documento de diseño del proyecto

El estado global del servidor se almacena en la estructura de tipo **GlobalState**, que se guarda de manera global en el archivo *globalstate.c*. Esta almacena las estadísticas, configuración de las transformaciones, tamaño máximo de los buffers y la información de los clientes.

En la estructura de tipo **Args** en *args.c* se guardan los argumentos que recibe el servidor. Los más destacables son las lista de usuarios, que serán consultadas directamente desde ambos protocolos cuando se realiza un intento de autenticación.

Para implementar el servidor de manera no bloqueante se decidió hacer uso del **selector** provisto por la cátedra, que despierta los file descriptors a medida que puedan realizar sus operaciones de lectura o escritura.

En el archivo *main.c* se inicializan los sockets pasivos respectivos de ambos protocolos, el selector, los handlers de las señales, y se procesan los argumentos recibidos. Las peticiones de conexión son atendidas por el socket pasivo, y su handler se encuentra en el archivo *protocols.c*.

Cuando un cliente inicia una conexión con el servidor, se crea una estructura **Client** asociada, tanto para POP3 como para DAJT. Esta contiene la información del estado del mismo, entre los que se encuentra:

- Una estructura de tipo **SocketData**, que contiene el file descriptor del socket del cliente, con buffers de lectura y escritura.
- Una máquina de estados asociada (usando la API stm provista por la cátedra).
- Un parser de comandos.
- Una estructura **CommandState** que almacena el estado del comando actual.
- El usuario, en caso de autenticación.
- El estado POP3 en el que se encuentra el usuario (AUTHENTICATION, TRANSACTION, UPDATE).
- Una estructura **ClientData**, que contiene información particular del protocolo POP3, se encuentra en NULL si es un cliente DAJT.

Como se mencionó, para manejar los distintos estados de los clientes, se utilizó la máquina de estados provista por la cátedra, que permite llamar distintos handlers para los file descriptors del selector, según en qué estado se encuentra el cliente. Los estados utilizados fueron WELCOME, COMMAND_READ, COMMAND_WRITE, ERROR, DONE.

Para procesar los comandos, se empleó un parser autómata de estados provisto por la cátedra, que puede recibir de a un caracter. Este carga gradualmente la información de un comando (nombre y argumentos) y lo finaliza cuando se detecta un salto de línea CRLF.

Cuando el cliente se encuentra en el estado COMMAND_READ, éste procesa la entrada del buffer de lectura del socket, usando el parser. Cuando detecta que el comando está listo para ejecutar, pasa al estado de COMMAND_WRITE o ERROR si hubo un error.

En el estado COMMAND_WRITE se llama al handler del comando actualmente almacenado en Client. Este inicializa los recursos necesarios, hace el procesamiento específico del comando (puede implicar escrituras sobre el buffer de escritura), y decide si este comando está finalizado. Fuera del handler del comando, se consume el buffer de escritura del cliente, para mandarle su contenido al socket del cliente. En caso de que el comando esté finalizado, se seguirá procesando el buffer de lectura del socket del cliente, con el parser. Cuando el buffer de lectura está vacío, se entrará al estado COMMAND_READ.

El estado ERROR se encarga de pasar del buffer de escritura al socket del cliente los mensajes de error, cuando este termina, pasa al estado de COMMAND_READ.

El estado DONE se utiliza para el comando QUIT y errores irrecuperables del cliente, donde se liberan los recursos de este.

La lógica exclusiva al protocolo POP3 se encuentra en el archivo **pop3.c**, donde se encuentran los handlers de sus comandos y de sus estados del autómata. Se usa la estructura **ClientData**, que almacena información necesaria para el manejo de los mails, entre los que se destacan:

- Una lista de los mails del usuario.
- File descriptor del mail actual.
- File descriptors de los pipes del transformador.
- Buffer del mail (para realizar *byte stuffing*)
- Variables del estado de los comandos multi-line.

Los comandos que no son multi-line se realizan en una llamada del handler, mientras que LIST y RETR pueden requerir varias pasadas para finalizarse, necesitando que se almacenen en client_data los progresos realizados. El comando LIST es manejado en su totalidad por su handler, mientras que el comando RETR tiene un tratamiento especial.

Como se mencionó en secciones pasadas, se tomó la decisión de realizar todas las operaciones de todos los file descriptors asociados a un cliente, en una única llamada del selector, sin distinguir cuál de los file descriptors fue elegido por el selector. Esto requirió un manejo de lectura con errores EAGAIN o EWOULDBLOCK, y de seteo de intereses complejo.

El flujo de datos para RETR sigue el esquema de la figura 1. Si las transformaciones están habilitadas, se lee del archivo del mail e intenta escribir sobre el pipe de entrada al transformador. En caso de que no haya podido escribir todo lo leído, se llama a la función lseek, para deshacer la lectura hasta donde fue escrito, se desactiva el interés de lectura del mail y se activa el interés de escritura del pipe. Cuando sea seleccionado el pipe, este hace una lectura del mail, y reactiva el interés de lectura del mail y desactiva el interés de escritura del pipe. La lectura del pipe de salida del transformador con transformaciones activadas, y la del archivo de mail cuando las transformaciones no están activadas, se manejan de la misma manera. Se considerará como "mail procesado" para referirse a ambos file descriptors en los casos respectivos. Cuando el mail procesado puede ser leído y mail_buffer puede ser escrito, se hace la lectura del mail procesado y se escribe sobre el mail_buffer. En esta lectura la función *mail_buffer_to_client_buffer* se ocupa de hacer el byte stuffing.

La función *mail_buffer_to_client_buffer* busca procesar los datos almacenados en el buffer *mail_buffer*. Primero, se busca la primera ocurrencia de un '\n'.

Dependiendo si este caracter se encuentra o no, se procede de una forma distinta: Si se encuentra un '\n', se copia el contenido del *mail_buffer* al buffer de escritura del socket, hasta antes del '\n'. Si el salto de línea fue el último caracter escrito, se debe esperar a que se escriba más contenido en *mail_buffer*. Este tiene un caso excepcional cuando se trata del '\n' final del mail, y se maneja de forma especial cuando el mail procesado se termina de leer.

Si el salto de línea está seguida de un punto, se realiza el bytestuffing, agregándose un punto extra al buffer.

Si no se encuentra un '\n', se copia el resto del *mail_buffer* en el buffer de escritura del socket y se actualiza la variable *last_carriage_return* según el último carácter copiado y se avanza el puntero de lectura.

En todos los casos, si se encuentra un '\n', se coloca un '\r', en caso de que no exista. Esta idea, inspirada en Dovecot, requiere constante llevar registro del último '\r', si fue parte del mismo read o si fue el último caracter del read anterior.

main.c main.cmain.c main.c main.c El protocolo DAJT se implementó en el archivo **dajt.c** de una manera que sea muy parecida a la de POP3 para poder reutilizar lógica ya implementada. Este

utiliza variables globales definidas en la estructura de tipo GlobalState (que es actualizada dentro de POP3) para poder retornar las estadísticas y/o cambiar las configuraciones ya nombradas.

Bibliografía

[1] John G. Myers y Marshall T. Rose. *Post Office Protocol - Version 3*. RFC. [Online; accedido 23/11/2023]. Dover Beach Consulting, Inc., 1996. URL: https://datatracker.ietf.org/doc/html/rfc1939.

[2] R. Gellens, C. Newman y L. Lundblade. *POP3 Extension Mechanism*. RFC. [Online; accedido 23/11/2023]. Qualcomm, 1998. URL: https://datatracker.ietf.org/doc/html/rfc2449.

[3] *The Open Group Base Specifications Issue 6*. https://pubs.opengroup.org/onlinepubs/009695399/basedefs/sys/select.h.html. [Online; accedido 23/11/2023]. 2000.

[4] *curl.1 the man page*. https://curl.se/docs/manpage.html. [Online; accedido 23/11/2023]. 2023-10-11 version 8.4.

[5] *lseek - reposition read/write file offset*. https://man7.org/linux/man-pages/man2/lseek.2.html. [Online; accedido 23/11/2023]. 2023.

[6] BlazeMeter Inc. *Concurrency Thread Group*. https://jmeter-plugins.org/wiki/ConcurrencyThreadGroup/. [Online; accedido 23/11/2023].

Apéndice

A. stress.js

```
const Pop3Command = require('node-pop3');
const clients = 500;
let finished = 0, totaltime = 0;

async function sleep(ms) {
        return new Promise(resolve => setTimeout(() => resolve(), ms));
}

function parseNanoToMSeconds(hrtime) {
        var seconds = (hrtime[0] + (hrtime[1] / 1e6));
        return seconds;
}

async function getEmail(index) {
        try {

            const pop3 = new Pop3Command({ user: "user", password: "pass",
            ↪  host: "localhost", port: 1110 });
            const start = process.hrtime();
            await pop3.connect();
            finished += 1;
            await pop3.command('USER', "user");
            await pop3.command('PASS', "pass");
            await pop3.command('RETR', 9);
            await sleep(2 * 1000);
            await pop3.command('QUIT');
            await sleep(2 * 1000);
            const seconds = parseNanoToMSeconds(process.hrtime(start));
            totaltime += seconds;
            console.log(totaltime / finished);
            return seconds;
        } catch (e) {
            console.log(`Error: ${e.message} - ${index}`);
        }
}

async function start() {
        let connections;
        try {
            connections = parseInt(process.argv.at(2), 10);
        } catch (e) {
            connections = clients;
        }
        const users = [];
        for (let i = 0; i < connections; i += 1)
            users.push(getEmail(i));
        await Promise.all(users);
}

start();
```