



Protocolos de Comunicación - 72.07

Trabajo Práctico Especial

Informe

Profesores:

Codagnone, Juan Francisco

Garberoglio, Marcelo Fabio

Kulesz, Sebastian

Grupo 9:

Francois, Gaston Ariel (62500)

Mentasti, José Rodolfo (62248)

Preiti Tasat, Axel Facundo (62618)

Suarez Durrels, Nicolas (62468)

ÍNDICE

| | |
|--|-----------|
| 1. Introducción..... | 3 |
| 2. Descripción de los protocolos y aplicaciones desarrolladas..... | 3 |
| 2.1 POP3..... | 3 |
| 2.2 PROTOS..... | 4 |
| 3. Problemas encontrados durante el diseño y la implementación..... | 6 |
| 3.1 POP3..... | 6 |
| 4. Limitaciones de la aplicación..... | 7 |
| 4.1 POP3..... | 7 |
| 4.2 PROTOS..... | 7 |
| 5. Posibles extensiones..... | 8 |
| 5.1 POP3..... | 8 |
| 5.2 PROTOS..... | 8 |
| 6. Conclusiones..... | 8 |
| 7. Ejemplos de prueba..... | 9 |
| 8. Guía de instalación..... | 17 |
| 9. Ejemplos de configuración y monitoreo..... | 18 |
| 10. Documentación de diseño del proyecto..... | 19 |
| 11. Referencias..... | 22 |

1. Introducción

En este informe se detalla acerca de la implementación de un servidor POP3 desarrollado por el grupo, respetando las reglas y convenciones establecidas por los RFCs 1939¹ y 2449².

Además, en el documento se expondrá el diseño del protocolo de monitoreo creado por los integrantes, al igual que su puesta en práctica mediante la implementación de las aplicaciones servidor y cliente del mismo.

2. Descripción de los protocolos y aplicaciones desarrolladas

2.1 POP3

El protocolo POP3 es un protocolo de aplicación, orientado a la conexión y confiable, diseñado para la descarga de mails desde un host servidor hacia un host cliente.

Se caracteriza por establecer una sesión entre el servidor y el cliente, durante la cual se transiciona entre tres estados: la autenticación, la transacción y la actualización. Cuando el cliente se conecta al servidor, este necesita ingresar sus credenciales para poder acceder a su casilla de correo. Si bien existen diferentes maneras de autenticarse, como los comandos **APOP** (RFC 1939, Sección 7, página 15¹) y **AUTH** (RFC 1734³), la opción tradicional es el par de comandos **USER** y **PASS**, en los cuales se debe indicar el nombre de usuario y la contraseña de la casilla de correo. Una vez autorizado, comienza el estado de transacción. Durante el mismo, el cliente podrá obtener información de su casilla de correo y descargar, listar y eliminar mails. Cuando el cliente desee finalizar la sesión, se transiciona al estado de actualización, durante el cual se eliminan aquellos mails que decidió borrar el cliente durante el estado de transacción, y finalmente la conexión se cierra. En el caso que decida finalizar la sesión antes de autorizarse, no se llega al estado de actualización y la conexión finaliza inmediatamente.

Según el RFC 1939¹, para que una aplicación servidor pueda ser considerado un servidor POP3, debe por lo menos soportar los comandos **QUIT**, **LIST**, **STAT**, **RETR**, **DELE**, **RSET**, **NOOP** y al menos un mecanismo de autenticación (generalmente, los comandos **USER** y **PASS**), los cuales se detallan en dicho documento. Además, tiene la capacidad para ser extendido a nuevos comandos y capacidades, como la técnica del pipelining, la cual permite a una aplicación cliente enviar una petición cuando aún no recibió una respuesta de la petición anterior.

El servidor implementado responde a los comandos mínimos, utiliza el mecanismo de autenticación **USER-PASS** y soporta pipelining. Cabe destacar que dicha implementación no es bloqueante y se ejecuta en un único thread, aprovechando la potencia de la función **select**⁴. El servidor también cuenta con un sistema de logging a archivo, en el cual se detalla la fecha y el nivel de importancia de cada log que escribe la aplicación. Como se detalla más adelante, el servidor exige indicar por argumentos (según los lineamientos de IEEE Std 1003.1-2008, 2016 Edition / Base definitions / 12. Utility Conventions⁵) los usuarios con sus correspondientes contraseñas y la ubicación del directorio Maildir. Opcionalmente, también se puede indicar el número de puerto por el cual se ejecutará, la cantidad máxima de mails a los que puede acceder un usuario en su casilla y el nivel mínimo de logging; sus valores por defecto son **1100**, **20** e **INFO**, respectivamente.

2.2 PROTO

Sumado a lo anterior, se implementó un servicio de monitoreo y configuración, el cual permite averiguar ciertas estadísticas y cambiar propiedades del servidor POP3. Este servicio respeta el protocolo definido en el RFC que acompaña esta entrega.

Para facilitar su uso, también se desarrolló una aplicación cliente, la cual recibe los comandos como argumentos, se encarga de armar el formato de cada petición según el RFC y finalmente, al recibir una respuesta del servidor, imprime el resultado por salida estándar.

Consultando el RFC, se puede averiguar que este protocolo opera sobre el protocolo de transporte UDP. Por lo tanto, como no existe una “sesión” entre el cliente y el servidor, para evitar que la aplicación cliente se detenga indefinidamente debido a la falta de respuesta por parte del servidor, se implementó un timer que comienza a ejecutarse cuando se envía una petición. Si la respuesta no llega en un cierto tiempo definido, la aplicación cliente deja de esperar al servidor y lanza un mensaje de error por *timeout* al usuario.

En esta primera versión, el servidor responde a las siguientes peticiones:

- Agregar un usuario al servidor
- Modificar la cantidad máxima de mails que puede ver un usuario al conectarse al servidor POP3
- Consultar la cantidad máxima actual de mails que puede ver un usuario al conectarse al servidor POP3
- Modificar la ubicación del directorio Maildir
- Consultar la ubicación actual del directorio Maildir
- Consultar la cantidad de conexiones que aceptó el servidor POP3 desde que empezó a ejecutarse
- Consultar la cantidad de clientes conectados en el momento al servidor POP3
- Consultar la cantidad de bytes que transfirió el servidor POP3

Todos estas operaciones están soportadas por la aplicación cliente, cada una con su respectiva opción de parámetro (por ejemplo, para consultar la cantidad de bytes transferidos se debe utilizar la opción **-b**).

Dado que el servicio modifica aspectos sensibles del servidor, es necesario contar con un sistema de autenticación. Para ello, en cada datagrama se debe incluir un *token*, una cadena de caracteres particular que el administrador debe conocer para que el servidor acepte la petición. Ese token tiene un valor por defecto, aunque puede modificarse antes de correr el servidor.

Además, otras de las capacidades con las que cuenta el servidor es el pipelining, lo cual permite a la aplicación cliente enviar todas las peticiones del cliente sin tener que esperar por la respuesta de una petición para poder enviar la siguiente. Esto permite agilizar la comunicación entre ambos hosts.

3. Problemas encontrados durante el diseño y la implementación

3.1 POP3

En cuanto al servidor POP3, para facilitar el acceso a los archivos del directorio Maildir, la aplicación utiliza las funciones `fstatat6` y `openat9` que permiten acceder a ellos utilizando un path relativo. Sin embargo, para ello fue necesario agregar el siguiente flag en la compilación: `-D_POSIX_C_SOURCE=200809L`. Se decidió hacer esto para no guardar para cada mail el path absoluto que lo accede y guardar únicamente su nombre y el path al directorio una única vez para la conexión. Si bien esto implica abrir el directorio cuando se hace un `RETR`, solo se hace momentáneamente para abrir el archivo y no deja un *file descriptor* abierto.

Por otra parte, se nos presentó un problema de eficiencia al realizar *byte stuffing*. Inicialmente, optamos por usar un parser que alertara al socket activo al momento de consumir `\r\n`. Sin embargo, esto generaba una gran caída en el rendimiento del servidor debido a las múltiples llamadas a funciones que realizaba por cada carácter consumido. Por lo tanto, se decidió reemplazar dicho parser por un conjunto de enteros (definidos en un `enum`) que manejan dicha lógica, evitando tener que llamar a otras funciones auxiliares. Esta decisión vino acompañada de una gran mejora en la velocidad de respuesta y una importante reducción en el tiempo de procesamiento del contenido.

4. Limitaciones de la aplicación

4.1 POP3

El servidor POP3 presenta principalmente tres limitaciones.

La primera de ellas es la cantidad de mails a las que puede acceder un usuario de su casilla. La implementación no cuenta con una lógica de acceso a directorio ilimitado. En cambio, el servidor solo otorga **DEFAULT_MAX_MAILS** (o el valor que haya definido algún administrador mediante PROTOS, o el valor definido por argumento al iniciar el servidor) mails por usuario.

La segunda es el manejo de la terminación de una respuesta al comando **RETR**. Luego de enviar el contenido del mail, el servidor termina la respuesta colocando incondicionalmente `\r\n.\r\n` al final. Por lo tanto, si el mail termina con `\r\n`, la respuesta del servidor va a finalizar con los bytes `\r\n\r\n.\r\n`, es decir que no verifica si el mail termina con `\r\n` y, si se cumple, solo agregar los últimos tres bytes. Se decidió seguir con esta lógica luego de las consultas realizadas en las clases, ya que resultaba ser la manera más confiable de asegurarse de que ningún cliente se quede “colgado”, esperando por la respuesta.

La tercera limitación que consideramos necesaria mencionar es que si un archivo de la carpeta de Maildir es eliminado por un usuario mientras otro mantiene una conexión con el servidor (por ejemplo, usando el comando **rm**), el servidor abortará si el cliente desea obtener dicho archivo. Si bien en un momento nos pareció que puede ser excesivo, consideramos a los archivos como un estado interno de la conexión, y un cambio en la ejecución del mismo puede resultar en un comportamiento indefinido.

Se puede hablar de una cuarta limitación que tiene que ver con el parseo de los comandos enviados por el cliente. El parser implementado interpreta el fin de un comando solamente si termina con `\r\n`. Es decir que el usuario siempre debe recordar agregar el parámetro **-C** al utilizar *netcat*^L, caso contrario el servidor no responderá al leer solamente `\n`. Sin embargo, no lo consideramos un caso importante ya que el RFC 1939 exige que los comandos terminen con `\r\n` (aunque algunas implementaciones soportan este caso).

4.2 PROTOS

Dado que el servicio opera sobre UDP, el protocolo no es confiable, por lo que no es posible garantizar que las peticiones son recibidas por el servidor. Tampoco se encarga de reenviar las peticiones si es que no recibe una respuesta. En consecuencia, el usuario está obligado a reescribir el comando en caso de obtener un error.

También, el protocolo no especifica una manera de tratar a las respuestas que no entran en un datagrama. Si bien las funcionalidades que ofrecemos no lo necesitaban, puede ser un problema al agregar funcionalidades futuras que deseen enviar una gran cantidad de datos en las peticiones o en las respuestas.

5. Posibles extensiones

5.1 POP3

La implementación puede extenderse para responder a peticiones de los comandos opcionales indicados en el RFC 1939, como **TOP**, aceptar otros mecanismos de autenticación, como **APOP**, y soportar las extensiones del protocolo mencionadas en el RFC 2449, como los código de respuesta (**RESP-CODES**, Sección 6.4, página 8).

Otras extensiones, a nivel implementación, son incorporar un sistema de paginación a las casillas de correo, de manera que un usuario pueda acceder a cualquier mail en su casilla; manejar el caso de que el contenido de un archivo termine en **\r\n**, el cual fue mencionado en la sección de limitaciones; y manejar el filesystem Mailbox⁸.

5.2 PROTOS

Una posible extensión que se podría aplicar sobre el protocolo de monitoreo y configuración es contar con sistemas de autenticación más seguros, los cuales pueden ir de un simple **USER-PASS**, como el que utiliza el protocolo POP3, hasta el uso de certificados TLS o SSL y claves asimétricas.

También se podría extender para que opere sobre el protocolo de transporte TCP o que pueda manejar otro tipo de operaciones, como el cambio de contraseña que finalmente no pudo ser implementado en esta versión.

6. Conclusiones

Este trabajo práctico nos permitió poner en práctica los conceptos aprendidos durante este cuatrimestre: la comunicación entre clientes y servidor, el funcionamiento de los protocolos de aplicación y transporte, la comprensión de documentos RFC, el manejo de sockets pasivos y activos y el cumplimiento de los requerimientos impuestos por un protocolo, entre otros.

Además, nos demostró la importancia de desarrollar teniendo en cuenta la posibilidad de que el mismo pueda ser extendido y contar con más capacidades en el futuro.

7. Ejemplos de prueba

Se buscó probar al servidor en varios casos mencionados en las clases de consulta. Estas pruebas se realizaron en el servidor **pampero** y en una máquina virtual con Ubuntu¹⁶. Esto se debe a que en algunos casos los límites impuestos por pampero para el uso del CPU no reflejaban lo que pasaba realmente en el servidor (y además llevaba a tiempos muy distintos dependiendo del nivel de uso que tenía en el momento de correr las pruebas).

Se utilizaron 2 MUA con el servidor desarrollado: **curl**¹⁰ y **Mozilla Thunderbird**¹¹. El primero se utilizó sobre todo en pruebas de stress que se mencionan a continuación, y el segundo para acceder a un correo que enviamos en las prácticas de mail anteriormente (con SMTP y Postfix¹⁷). Para Thunderbird, agregamos la siguiente configuración.

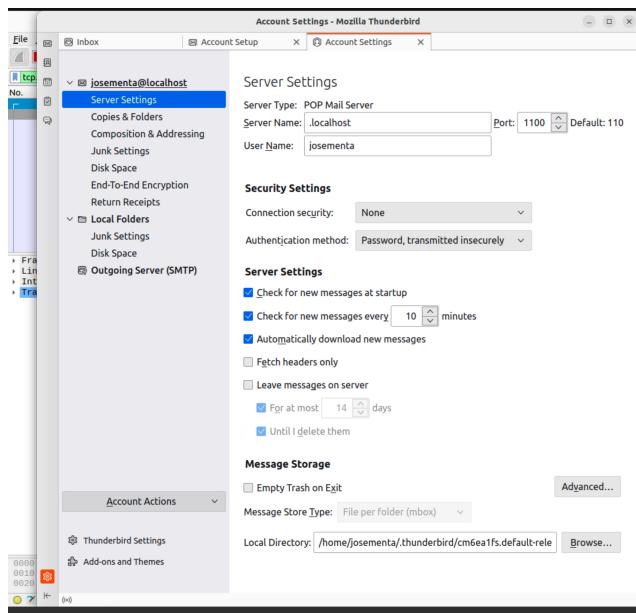


Imagen 7.1: Configuración de Thunderbird.

En un principio, lo testeamos con un correo de prueba (que no seguía el formato estándar) y vimos la siguiente interacción.

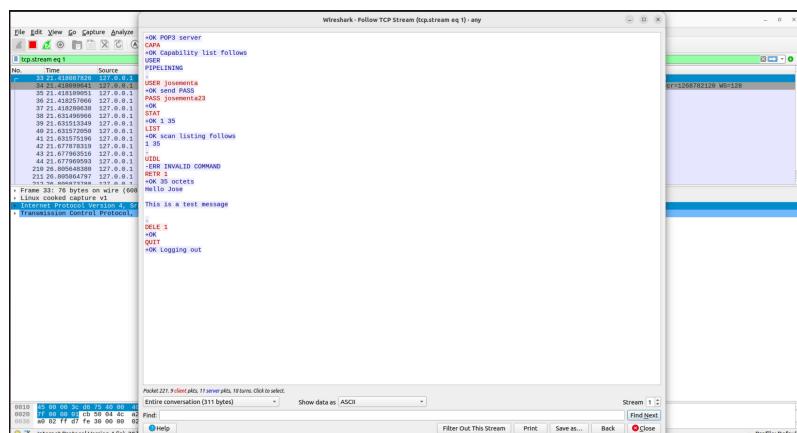


Imagen 7.2: Captura Wireshark interacción entre Thunderbird y el servidor

Luego, mandamos un correo generado cuando practicamos SMTP con Postfix.

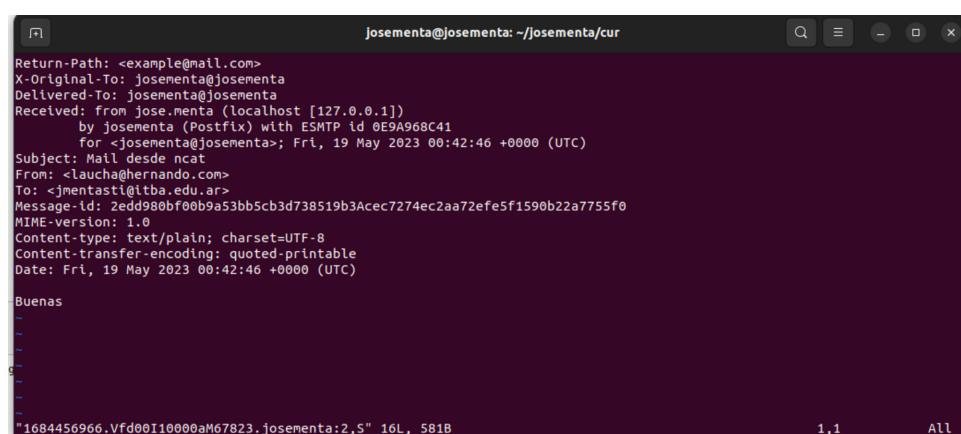


Imagen 7.3: Contenido de email de prueba.

```

Jun 18 00:26:11 Wireshark - Follow TCP Stream (tcp.stream eq 0) - any
+OK POP3 server
CAPA
+OK Capability list follows
USER
PIPELINING
.
USER josementa
+OK send PASS
PASS josementa23
+OK
STAT
+OK 1 581
LIST
+OK scan listing follows
1 581
.
UIDL
-ERR INVALID COMMAND
RETR 1
+OK 581 octets
Return-Path: <example@mail.com>
X-Original-To: josementa@josementa
Delivered-To: josementa@josementa
Received: from jose.menta (localhost [127.0.0.1])
        by josementa (Postfix) with ESMTP id 0E9A968C41
        for <josementa@josementa>; Fri, 19 May 2023 00:42:46 +0000 (UTC)
Subject: Mail desde ncat
From: laucha@hernando.com
To: jmentasti@itba.edu.ar
Message-ID: 2a1d990f0b9a53bb5cb3d738519b3acec2aa72eef5f1590b22a7755f0
MIME-version: 1.0
Content-type: text/plain; charset=UTF-8
Content-transfer-encoding: quoted-printable
Date: Fri, 19 May 2023 00:42:46 +0000 (UTC)

Buenas

.
DELE 1
+OK
QUIT
+OK Logging out

```

Packet 188. 9 client pkts, 11 server pkts, 18 turns. Click to select.

Entire conversation (860 bytes) Show data as ASCII Stream 0 Find Next Filter Out This Stream Print Save as... Back Close Help

Imagen 7.4: Captura de Wireshark con email de prueba.

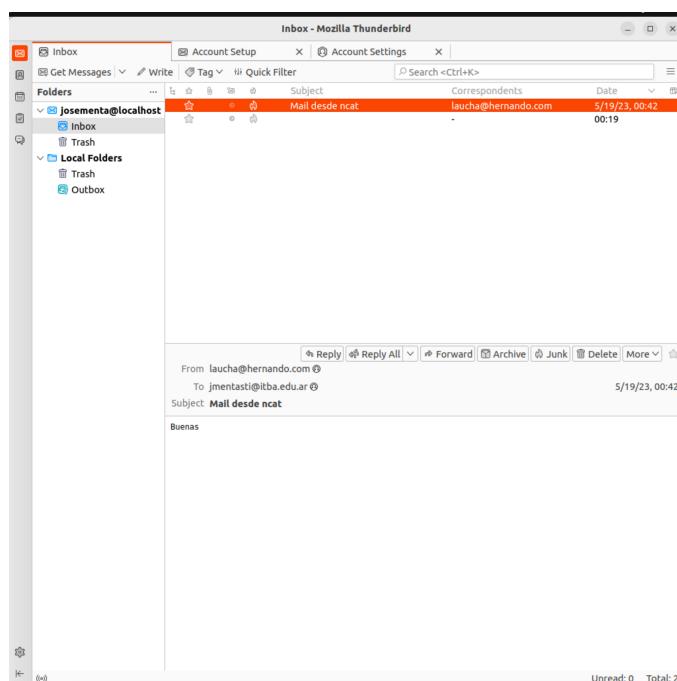


Imagen 7.5: Vista de thunderbird del mail

Podemos ver que ahora el MUA obtiene correctamente los campos, como el asunto, para mostrarlos en su UI.

Otro testeo que se hizo fue el del tamaño de los buffers que debíamos usar. Para ello (y otros testeos) se creó un archivo grande (a veces de 1GB, a veces de 500MB, o de 100MB). Esto se hizo con los siguientes comandos.

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

Luego, se inició la descarga de alguno de estos archivos con nuestro servidor (**RETR**), usando *curl*.

```
curl pop3://user:password@localhost:port/num > /dev/null
```

Descargando siempre el mismo archivo pero cambiando los tamaños de buffer, notamos 2 cosas:

- A partir de los 8000 bytes ya no se observaba un cambio significante (se llegó a testear con 32.000).
- Tamaños menores a los 2000 bytes resultaban en que el servidor ocupe mucho tiempo en entrada y salida, provocando que el uso de CPU sea bajo.

Entonces, se decidió seguir con buffers de tamaño de 4096 bytes.

Otra prueba realizada fue la de comparar nuestra velocidad con la de Dovecot¹². Si bien entendíamos que resultaba difícil que lo igualáramos, lo usamos como una referencia para nuestros tiempos. En esta comparación, se midieron los tiempos para traer el archivo de 1GB.

En un principio, nuestros tiempos eran aproximadamente 10 veces más lentos que los de Dovecot. Esta comparación fue la que nos llevó a pensar en los problemas que tenía nuestro mecanismo de *byte stuffing* con un parser (pues es lo que más se ejecuta cuando se transfiere un mensaje), y también nos dio una buena razón para probar las optimizaciones que nos ofrecen los compiladores (agregamos el flag **-O3**). Finalmente, llegamos a tiempos de entre 2 a 3 veces más lentos que Dovecot, que si bien parecen bastante todavía, nos demostró que los cambios realizados tuvieron efecto.

Siguiendo con la prueba de descargar el archivo grande, se observó el comportamiento del servidor cuando un cliente cierra la conexión en la mitad de la descarga, observando con **strace** que el servidor deje de suscribirse para esa conexión y libere los recursos asociados.

Otra prueba fue intentar “medir” cómo varían los tiempos cuando hay varios usuarios obteniendo datos en un mismo momento. Si bien no fue una prueba rigurosa, para testearlo se levantaron 4 terminales, cada una usando *curl* para acceder a un usuario distinto y descargar un archivo lo suficientemente grande como para no terminar antes de que el resto empiece. Con las 4 conexiones en simultáneo, se llegó al siguiente tiempo.

```

M See 'snap info <snapname>' for additional versions.
M josementa@josementa:~/rodo/cur$ curl pop3://rodo:menta@localhost:1100/
M 1 331991579
josementa@josementa:~/rodo/cur$ curl pop3://rodo:menta@localhost:1100/1 > /dev/null
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
          Dload  Upload   Total   Spent   Left  Speed
100  316M    0  316M    0      0  14.4M       0  --::--  0:00:21  --::-- 14.5M
josementa@josementa:~/rodo/cur$ █

```

Imagen 7.6: Tiempo en recibir email via curl con 4 conexiones simultáneas

Y corriendo con un único usuario, se obtiene lo siguiente.

```

josementa@josementa:~/rodo/cur$ curl pop3://rodo:menta@localhost:1100/1 > /dev/null
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
          Dload  Upload   Total   Spent   Left  Speed
100  316M    0  316M    0      0  82.2M       0  --::--  0:00:03  --::-- 82.2M
josementa@josementa:~/rodo/cur$ █

```

Imagen 7.7: Tiempo en recibir email via curl con una conexión.

Si bien los datos no pueden llevar a una conclusión certera (pues se necesitaría una medición más precisa), inferimos que el orden de los tiempos parece el correcto.

Para probar la capacidad de pipelining, se redirigió la salida estándar de `printf` a `ncat`, esperando que se ejecuten todos los comandos enviados.

```

josementa@josementa:~$ printf "user jose\npass menta\nlist \n" | ncat localhost 1100 -c
+OK POP3 server
+OK send PASS
+OK
+OK scan listing follows
1 50
2 1383298249
.
josementa@josementa:~$ █

```

Imagen 7.8: Ejemplo de uso con pipelining

Para probar las lecturas parciales, se utilizó el comando `stty` para que netcat envíe paquetes TCP antes de la llegada del salto de línea.

```

stty -icanon && nc -C 192.168.122.99 110

```

Y observamos un comportamiento normal del servidor.

Para las escrituras parciales, se limitó en el código a que todos los llamados a las funciones `send` envíen como máximo 1 byte. Si bien la performance empeoró, por cuestiones únicamente de testeо, se probó traer un archivo relativamente grande y ver que se obtuviera la información correcta.

Además, se buscó verificar que el servidor soporte 500 conexiones, al menos para el caso donde ninguna interactúa con el servidor. En estos casos, buscamos observar 2 cosas:

- Que el uso del CPU del servidor no se mantenga alto, ya que la no interacción de las conexiones debería provocar que el mismo se mantenga bloqueado (al menos, hasta el timeout indicado en el selector).
 - Que el cierre de las conexiones haga que el servidor libere sus recursos asociados (que nuevamente se siguió con `strace`)

```
top - 11:53:35 up 9y 4d, 20:16, 10 users, load average: 0.00, 0.00, 0.00
Tasks: 497 total, 1 running, 484 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.2 us, 0.2 sy, 0.0 ni, 99.6 id, 0.0 wa, 0.0 hi, 0.1 si, 0.0 st
Mem: 1976.2 total, 789.1 free, 446.4 used, 818.6 buff/cache
Mib Swap: 2847.0 total, 1408.6 free, 642.4 used, 1299.5 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
3186455 jmentasti 20 0 10102 4732 3308 R 1.0 0.2 0:00.32 top
3186450 root 20 0 0 0 0 I 0.3 0.0 0:00.05 kworker*
1 root 20 0 174408 7348 4768 S 0.0 0.4 8:19.29 system
2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
3 root 20 0 0 0 0 S 0.0 0.0 0:00.00 rcu_bh
4 root 0 -20 0 0 0 0 I 0.0 0.0 0:00.00 rcu_par-
5 root 0 -20 0 0 0 0 I 0.0 0.0 0:00.00 slab_flt*
6 root 0 -20 0 0 0 0 I 0.0 0.0 0:00.00 netns
7 root 0 -20 0 0 0 0 S 0.0 0.0 0:00.00 ksoftirq-
10 root 0 -20 0 0 0 0 I 0.0 0.0 0:00.00 mm_perc-
12 root 20 0 0 0 0 S 0.0 0.0 0:00.00 rcu_tas*
13 root 20 0 0 0 0 S 0.0 0.0 0:00.00 rcu_tas*
14 root 20 0 0 0 0 S 0.0 0.0 0:13.84 ksoftir-
15 root 20 0 0 0 0 S 0.0 0.0 0:00.00 rcu_tas*
16 root 11 0 0 0 0 S 0.0 0.0 0:00.00 rcu_tas*
17 root -51 0 0 0 0 S 0.0 0.0 0:00.00 idle_in+
19 root 20 0 0 0 0 S 0.0 0.0 0:00.00 cpuhp/0
```

Imagen 7.9: Ejemplo de prueba con 500 conexiones en simultáneo

Imagen 7.10: Ejemplo de cierre de las 500 conexiones

Otra cuestión que nos parece interesante mostrar es un problema que surgió cuando se comparaba el *byte stuffing* del servidor con Dovecot. Se pudo ver un caso donde Dovecot no lograba mantener el contenido original del “mail”

Imagen 7.11: Comparación del resultado entre dovecot y nuestro servidor

Esto se debía a que no escapaba el primer . de la primera línea.

```
josementa@josementa: ~$ ncat localhost 1100 -C
+OK POP3 server
user josementa
+OK send PASS
pass josementa23
+OK
list
+OK scan listing follows
1 18
.
retr 1
+OK 18 octets
..
..hola
..
fin
.

^C
josementa@josementa: ~$ ncat localhost 110 -C
+OK Dovecot (Ubuntu) ready.
user josementa
+OK
pass josementa23
+OK Logged in.
list
+OK 1 messages:
1 45
.
retr 1
+OK 45 octets
.
..hola
..
fin
.
^C
josementa@josementa: ~$
```

Imagen 7.12: Comparación del byte stuffing entre Dovecot y nuestro servidor

Si bien entendemos que posiblemente Dovecot no maneje estos casos por el formato de contenido que sirve, nos pareció interesante marcarlo.

También, vimos otra diferencia con respecto a la manera que tiene Dovecot de procesar el contenido antes de mandarlo, ya que este se encarga de reemplazar todas las apariciones de `\n` por `\r\n`. Al principio nos sorprendió, ya que al obtener las diferencias con lo que da nuestro servidor se veía algo en cada línea, pero después se encontró que dejando de lado esto (con el comando `dos2unix`) la única diferencia entre los dos resultados era la línea que nuestra implementación agrega por cuestiones ya mencionadas.

```
100 150M 0 150M 0 0 174M 0 174M 0 174M
josementa@josementa:~$ dos2unix dovecot_output
dos2unix: converting file dovecot_output to Unix format...
josementa@josementa:~$ diff -u dovecot_output my_output
--- dovecot_output 2023-06-18 01:18:52.102856039 +0000
+++ my_output 2023-06-18 01:18:05.363377577 +0000
@@ -1839606,3 +1839606,4 @@
 Vu89BBWU5B0jgXr4cdQCMltR0kLxVerrt52jlCeZ//RRQmn73oAl0U0AN+7nTe5QhrI/kkWu/FJf
 4mE3tBKOG3T0XmEIjCP1ZdfL9WMhInUETTVHxexFEqxXZauQtxpuS7/Nevu4ehQlmLVhGDTrrJ6N
 tQ==
+
josementa@josementa:~$
```

Imagen 7.13: Diferencia entre los resultados obtenidos

8. Guía de instalación

Para compilar el proyecto, primero se debe dirigir al directorio con los códigos fuente

```
cd <path_to_project>
```

Asegurándose de tener instalado [make¹³](#), se debe compilar el proyecto ejecutando

```
make all CC=<c_compiler>
```

Donde puede especificar a [gcc¹⁴](#) o [clang¹⁵](#) como compilador. Se advierte que pueden aparecer algunos warnings al compilar (que fueron analizados y se espera que sean falsos positivos).

Luego, en la carpeta **bin/** se encuentran los 2 archivos ejecutables: **popserver** y **popadmin**. El primero es el servidor POP3 y el segundo es el cliente de configuración y monitoreo que utiliza el protocolo PROTO.

Para correr el servidor, se deben indicar los siguientes argumentos:

- **-d <dir>**: path absoluto al directorio Maildir. El servidor buscará un subdirectorio para cada usuario (con su nombre) y los emails en la carpeta **/cur**. Es importante que el path esté terminado por **/**.

Opcionalmente, se pueden agregar usuarios, determinar el token utilizado por el cliente para hacer cambios, entre otros (se pueden ver todos agregando **-h**). Para no tener que contar con permisos de administrador, el servidor va a escuchar en el puerto 1100 para POP3 (este puerto es configurable) y en el 1024 para el monitoreo.

El servidor dejará los logs asociados a su actividad en la carpeta **log/**, que se ubicará en el directorio donde se ejecute el programa. Allí se podrán ver los accesos de usuarios registrados por el servidor, que se pueden determinar antes de correr el mismo o agregar en tiempo de ejecución con el cliente.

En el caso del cliente, se podrán indicar las operaciones que se desean realizar utilizando argumentos del programa. Para ver un listado de las mismas, se puede usar **-h**.

9. Ejemplos de configuración y monitoreo

El cliente de monitoreo y configuración permite obtener estadísticas del servidor y realizar cambios mientras este funciona.

Por ejemplo, para agregar el usuario **jose** con contraseña **menta**, se puede correr lo siguiente.

```
./bin/popadmin -A jose:menta
```

A su vez, para obtener datos como el maildir actual del servidor o el máximo de mails que muestra para cada usuario, se puede ejecutar lo siguiente.

```
./bin/popadmin -d -m
```

Si se quiere cambiar alguno de estos valores, como por ejemplo el directorio utilizado para buscar los mails, se puede correr lo siguiente.

```
./bin/popadmin -D /home/gfrancois/
```

Es importante notar que estos cambios se realizarán para todas las conexiones a partir del momento de su confirmación, sin cambiarse las conexiones que se encuentren abiertas.

Finalmente, para obtener las estadísticas, podemos ver los 3 valores (conexiones históricas, conexiones actuales y bytes transferidos) ejecutando

```
./bin/popadmin -p -c -b
```

Cada vez que se desee ejecutar al cliente, se deberá ingresar el token de administrador. Este tiene como valor **1234** por defecto, aunque puede ser cambiado antes de correr al servidor con el comando **-t**.

10. Documentación de diseño del proyecto

El proyecto se encuentra dividido en 2 partes: el servidor POP3 y el protocolo de monitoreo.

En cuanto al servidor, se buscó llegar a una implementación no bloqueante que utiliza un único thread para atender las consultas. Para ello, se utilizó la abstracción del **selector** junto con la **stm** y **buffer** para mantener los estados y los datos asociados a cada conexión, todas estas provistas por la cátedra.

En el archivo **main.c** se configura al servidor, recibiendo los argumentos de configuración donde se utilizan los archivos **args.c** para recibirlos y **usersADT.c** para almacenar a los usuarios (que no es estrictamente un ADT). Luego, se abren los sockets correspondientes para atender los pedidos de los clientes (tanto de POP3 como del administrador). Después de inicializar al **selector**, se agregan los sockets pasivos TCP de POP3 y el socket UDP de configuración al mismo, y se entra en un ciclo para atender conexiones y pedidos.

En el archivo **admin.c** se encuentra toda la lógica asociada a recibir y responder pedidos efectuados por el cliente de monitoreo. Dado que se utiliza UDP, ni bien se recibe un pedido, se lo atiende y envía sin pasar por el **selector** antes de escribir, por lo que se utiliza un único socket que se mantiene suscrito a lectura.

En cuanto al protocolo de comunicación establecido entre el cliente de monitoreo y el servidor POP3, se procedió a realizar un protocolo propio como se indicaba en el enunciado. Al utilizar UDP, se optó por armar un datagrama por solicitud al servidor. Por ejemplo, si se quiere agregar un usuario y obtener el número de bytes transferidos, entonces se enviará un datagrama con la solicitud de agregar un usuario y luego se enviará otro datagrama con la solicitud de obtener el número de bytes transferidos. En cuanto a los encabezados del protocolo, se buscó la menor cantidad de encabezados posibles con la idea de reducir la lógica de ambos sistemas y reducir los tiempos de parsers cuando no eran necesarios. En este sentido, se agregó un header de versión, para admitir posibles ampliaciones al protocolo.

En el archivo **pop3.c** se encuentra toda la implementación de los comandos del protocolo. Si bien esto no nos parece algo muy escalable (el archivo es realmente extenso), dado que casi todas las funciones se encuentran muy relacionadas entre sí (por ejemplo, todas manejan la misma estructura de estado de conexión), nos pareció que dividirlos iba a introducir confusión en el proyecto, ya que si bien era posible dividir a la implementación en varios archivos, el hecho de que todas las funciones usen el mismo estado hace que ninguna esté aislada de la otra.

Cada conexión es asociada a un *file descriptor* (el socket activo), que se registra en el **selector** junto con una estructura llamada **pop3**. En esta, se guardan los buffers asociados a lectura, escritura y archivos abiertos, junto con otras variables para mantener el estado de la conexión y datos asociados a ese estado (por ejemplo, el estado de la sesión POP3, que nos permite diferenciar a los comandos que puede realizar cada usuario).

Todos los comandos tienen información en común: un nombre, una función que verifica a los argumentos que recibe, y una función que ejecuta para producir su respuesta. Con esto, la lectura de todos los comandos se realiza en una única función, donde se lee lo ingresado en el socket activo y se lo guarda en un buffer para analizarlo a continuación. Pasando lo ingresado por un parser, se lee la entrada del usuario hasta el fin de la línea (`\r\n`) y se determina si es un comando válido o no. En ambos casos, se avanza el buffer hasta el fin de este comando (lo que nos permite utilizarlo como cola para varios comandos) y se suscribe a la escritura del socket.

En la escritura, lo primero que se hace es llamar a la función asociada al comando en cuestión, donde se va a completar en el buffer de escritura con la información correspondiente. En todos los casos, cuando estas funciones retornan, se puede escribir información en el socket. En la mayoría de los casos, al término de la función se escribe lo producido en el socket y se vuelve a suscribir a escritura si todavía no se terminó de generar la respuesta. Solo en algunos casos como el de error o el de `RETR` se deja a la función sin escribir, ya que se fue a buscar más información o se debe cerrar la conexión por un error no recuperable.

En el caso de los comandos que devuelven una línea, las funciones se encargan de actualizar los datos de la conexión y luego agregan su respuesta al buffer de salida, marcando como terminada la ejecución del comando. En el caso de `LIST`, cuando se llama para todos los archivos del directorio no siempre termina inmediatamente en la primera llamada, por lo que mantiene un estado interno para que cuando vuelva a ser llamada siga con la respuesta desde donde “la dejó la última vez”. En el caso de `RETR`, el *byte stuffing* se realiza en la función del comando y solo se abandona esta cuando se debe leer contenido del archivo (para lo que se utiliza otro estado, donde la máquina de estados tiene funciones asociadas que abren el archivo y se suscriben a lectura cuando llegan a este, y luego de leer y guardarlos en un buffer intermedio se suscriben a la escritura del socket para seguir generando la respuesta). Esto puede ser una limitación, ya que se puede decir que estamos desaprovechando una escritura en el socket, pero consideramos que dado que vamos a volver a escribir inmediatamente en la siguiente llamada, consideramos que no afecta a la performance gravemente.

El cierre de las conexiones se realiza cuando hay un error no recuperable para seguir la conexión. Cuando ocurre esto, se llama a la función que elimina a los *file descriptors* asociados a la misma del `selector`, y a su vez esto hace que se libere la estructura asociada a la conexión sólo la última vez que es referenciada (utilizando un contador de referencias).

En cuanto al administrador, la arquitectura es similar aunque más simple: en el archivo `main.c` se configuran las conexiones necesarias, en `admin_args.c` se reciben los argumentos ingresados por el usuario y se guarda un request al servidor por cada uno, y en `admin_resp` se reciben y parsean las respuestas del servidor ante esos pedidos. Para un mayor grado de detalle acerca de cómo es la comunicación entre el administrador y el servidor y qué protocolo se utiliza se recomienda leer el RFC provisto en la entrega del proyecto.

El cliente envía todos los pedidos al servidor, y luego va recibiendo los mismos asociándolos a cada pedido con un ID posibilitado por el protocolo. Al llegar la respuesta, se asocia el número de ID de la respuesta con la solicitud con el mismo ID y se imprime desde el cliente la respuesta obtenida. En el caso en que una respuesta no llegue, se espera una cantidad de tiempo predeterminada hasta que se le muestra un mensaje de error al usuario para que vuelva a intentar ejecutar el pedido (*timeout*).

11. Referencias

¹: *Post Office Protocol - Version 3*. J. Myers, M. Rose [May 1996].
<https://www.ietf.org/rfc/rfc1939.txt>

²: *POP3 Extension Mechanism*. R. Gellens, C. Newman, L. Lundblade [November 1998].
<https://datatracker.ietf.org/doc/html/rfc2449>

³: *POP3 AUTHentication command*. J. Myers [December 1994].
<https://datatracker.ietf.org/doc/html/rfc1734>

⁴: *select*. (s.f.). The Open Group Publications Catalog.
<https://pubs.opengroup.org/onlinepubs/7908799/xsh/select.html>

⁵: *The Open Group Base Specifications Issue 7, 2018 edition*. (s.f.). The Open Group Publications Catalog. <https://pubs.opengroup.org/onlinepubs/9699919799/nframe.html>

⁶: *fstatat*. (s.f.). The Open Group Publications Catalog.
<https://pubs.opengroup.org/onlinepubs/9699919799/functions/fstatat.html>

⁷: Contributors to Wikipedia projects. (2004, 6 de septiembre). *netcat - Wikipedia*. Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/Netcat>

⁸: Contributors to Wikipedia projects. (2002, 3 de agosto). *Mbox - Wikipedia*. Wikipedia, the free encyclopedia. <https://en.wikipedia.org/wiki/Mbox>

⁹: *The Open Group Base Specifications Issue 7*. (s.f.). The Open Group Publications Catalog.
<https://pubs.opengroup.org/onlinepubs/9699919799.2008edition/>

¹⁰: *curl*. (s.f.). curl. <https://curl.se/>

¹¹: *Thunderbird — Haz el correo más fácil*. (s.f.). Thunderbird.
<https://www.thunderbird.net/es-ES/>

¹²: *Dovecot | The Secure IMAP server*. (s.f.). Dovecot | The Secure IMAP server.
<https://www.dovecot.org/>

¹³: *Make - GNU Project - Free Software Foundation*. (s.f.). The GNU Operating System and the Free Software Movement. <https://www.gnu.org/software/make/>

¹⁴: *GCC, the GNU Compiler Collection- GNU Project*. (s.f.). GCC, the GNU Compiler Collection- GNU Project. <https://gcc.gnu.org/>

¹⁵: *Clang C Language Family Frontend for LLVM*. (s.f.). Clang C Language Family Frontend for LLVM. <https://clang.llvm.org/>

¹⁶: *Enterprise Open Source and Linux | Ubuntu*. (s.f.). Ubuntu. <https://ubuntu.com/>

¹⁷: *The Postfix Home Page*. (s.f.). The Postfix Home Page. <https://www.postfix.org/>