



Trabajo Práctico Especial Protocolos de Comunicación

Primer Cuatrimestre 2023

Pedro J. López Guzmán 60711

Martín E. Zahnd 60401

Docentes

Codagnone, Juan Francisco

Garberoglio, Marcelo Fabio

Kulesz, Sebastian

Grupo 2⁴

Índice

1. Abstracto	1
2. Protocolos y aplicaciones desarrolladas	2
2.1. Suposiciones	2
2.2. Componentes	2
2.2.1. Estado de Autorización	2
2.2.2. Estado de Transacción	3
2.2.3. Estado de Actualización	3
2.3. Estructura del proyecto	4
2.3.1. Parser	4
2.3.2. Flujo General de una conexión	4
2.3.3. Manejo de datos	5
2.3.4. Buffers	5
2.4. Protocolo de monitoreo	6
3. Pruebas de los protocolos	7
3.1. Integridad de los datos	7
3.2. Pipelining	8
3.3. Conexiones simultáneas	8
4. Problemas encontrados	10
4.1. Problemas menores	10
4.1.1. Makefile	10
4.1.2. Estilo de código	10
4.2. Envío del caracter terminador	10
4.3. Maildir	11
4.4. Envío de un caracter extraño	11
4.5. Lectura de archivos grandes	12
4.6. Uso de programa externo para lectura de archivo	13
4.7. Población de maildirs	13

5. Limitaciones de la aplicación	14
6. Posibles extensiones	15
7. Conclusiones	16

1. Abstracto

En este documento se presenta el desarrollo del Trabajo Práctico Especial para la asignatura de Protocolos de Comunicación. El objetivo del trabajo consistió en la implementación de un servidor que utilice el protocolo de mailing POP3 definido en el RFC 1939 [1]. , así como el diseño e implementación de un protocolo de monitoreo adicional. Este protocolo no solo cumple funciones de monitoreo, sino también de administración del servidor.

2. Protocolos y aplicaciones desarrolladas

Diseñamos y desarrollamos un servidor POP3 no bloqueante basado en el estándar RFC 1939 [1]. Este servidor es capaz de atender conexiones simultáneas tanto en IPv4 como en IPv6 sobre TCP, y puede manejar más de 500 conexiones al mismo tiempo sin bloquearse. Además, el servidor admite la técnica de PIPELINING y utiliza los comandos USER y PASS para la autenticación. Siguiendo el estándar, los comandos del protocolo no son sensibles al uso de mayúsculas o minúsculas.

Posteriormente, implementamos un protocolo de monitoreo y administración que permite consultar datos históricos del servidor, como la cantidad de bytes enviados y el número de conexiones. Para obtener más detalles sobre este protocolo, puedes consultar la sección [Protocolo de monitoreo](#).

2.1. Suposiciones

- Asumimos que el Maildir contiene correos electrónicos bien formados, todos ellos terminados con dos caracteres: un carriage return (**CR**) seguido por un line feed (**LF**), es decir, **CRLF**.
- Los correos electrónicos leídos por un cliente (utilizando el comando **LIST**) son movidos al subdirectorio **cur** del Maildir y a su nombre se le agrega el flag **S** según el formato Maildir [2].

2.2. Componentes

A continuación detallamos los distintos componentes implementados en el funcionamiento del servidor, que decisiones fueron tomadas y cuál es cada uno de sus roles.

2.2.1. Estado de Autorización

Una vez que se establece una conexión, el servidor POP3 envía inmediatamente un mensaje de bienvenida al cliente. Este mensaje comienza con un indicador positivo

(+OK), seguido de un espacio y un saludo.

Después del saludo de bienvenida, se ingresa al **Estado de Autorización** (Authorization State), donde el usuario debe identificarse para acceder a sus correos. En este estado, se utiliza un analizador (parser) para extraer el primer parámetro. Se toma todo lo que sigue después de un espacio como los parámetros del comando.

Es necesario que un cliente utilice primero el comando de **USER** seguido inmediatamente por el de **PASS**. En caso de que se ingrese un usuario inexistente, la respuesta en el primer comando es un indicador negativo **-ERR**. Si se intenta acceder a un usuario que ya se encuentra conectado, el servidor responde con un indicador negativo seguido con un mensaje detallando la razón. Dentro del Estado de Autorización también se encuentran disponibles los comandos de **QUIT** para terminar la sesión y de **CAPA** para indicar las capacidades que cumple el servidor.

2.2.2. Estado de Transacción

Una vez identificado el usuario, el cliente cambia su estado al **Estado de Transacción** (Transaction State). En este estado es en donde un usuario puede interactuar con sus correos. Para la implementación del guardado de correos seguimos el patrón **maildir** [2]. Dentro de este estado se encuentran disponibles múltiples comandos. Se decidió que al ejecutar el comando **RETR** el mail solicitado fuese movido de su maildir una vez terminada la sesión y se volviese inaccesible de nuevo a través de dicho comando.

2.2.3. Estado de Actualización

Al ejecutarse el comando **QUIT** en el Estado de Transacción, se transita hacia el **Estado de Actualización** (Update State) en el cual no se recibe ningún tipo de input por parte del cliente. En este estado se realizan las operaciones necesarias en el sistema de archivos según las interacciones que tuvo el cliente en el estado anterior. Si dichas operaciones pudieron realizarse sin inconvenientes se envía un mensaje con el indicador positivo. En el caso contrario, se envía un mensaje de error. Una vez terminado, se cierra la conexión

2.3. Estructura del proyecto

Utilizamos tres componentes provistos por la cátedra: `selector.c`, `buffer.c` y `stm.c`. El primero se encarga de la multiplexación de distintos *file descriptors* para gestionar la lectura y escritura de distintas conexiones. El segundo cumple el rol de un espacio de memoria en el cual se puede escribir y leer en distintos momentos y ofrece ventajas en el manejo de memoria. Por último, el tercero corresponde a una máquina de estados que nos permite modificar el comportamiento del cliente según los distintos estados definidos, así como abstraer los estados documentados en POP3.

2.3.1. Parser

Implementamos un módulo que contiene un parser el cual nos simplificaba la definición de estados y transiciones, permitiéndonos entender a fondo el proceso de lectura de paquetes entrantes. La novedad de dicho parser es la posibilidad de definir transiciones entre estados aclarando el complemento de un conjunto de caracteres o un rango entre ellos. Por ejemplo, suponga que se quiere pasar de un estado A a un estado B al leer cualquier caracter excepto por los caracteres "a b c".

2.3.2. Flujo General de una conexión

Entender el uso de la API provista por UNIX se nos presentó como un desafío interesante. Si bien cada parte del proyecto varía en su código (Consecuencia que naturalmente surgió al estar cada vez mas experimentados con las herramientas), todas se pueden resumir en un flujo como este:

```
1  leo del socket del cliente
2  parseo lo leído
3  si el parseo finalizó {
4      proceso los datos y genero respuesta
5      me suscribo a escritura para mandar la respuesta
6  }
7  sino {
8      me suscribo a lectura para leer de nuevo
9  }
```

Código 1: Flujo básico de lectura y escritura con cliente

Este flujo, presentado en la Figura 1, se presenta tanto en la implementación de POP3 como en la de monitoreo.

2.3.3. Manejo de datos

Como cada cliente posee sus propios datos y necesita guardarlos mientras está conectado, se decidió crear una estructura en la que se guardasen los datos imprescindibles para cada conexión. Entre estos datos están los buffer de entrada y salida, el nombre de usuario en caso de que la conexión haya pasado el estado de Autorización y otros datos relevantes para el buen funcionamiento del servidor. Esto aplica tanto a POP3 como al protocolo de monitoreo.

2.3.4. Buffers

Para la mayoría de respuestas generadas por POP3, no son necesarios demasiados bytes. Sin embargo, en el caso del comando **RETR**, este nos tiene que devolver un archivo correspondiente a un mail, y este archivo no posee límite en su tamaño. Para resolver esto decidimos hacer que cada cliente tenga un buffer de salida, el cual será leído en cada iteración de escritura para realizar un send en caso de que haya datos para leer. Decidimos que cada buffer tenga un tamaño de 32KB, lo cual nos permite almacenar grandes porciones de una respuesta sin ocupar demasiado espacio.

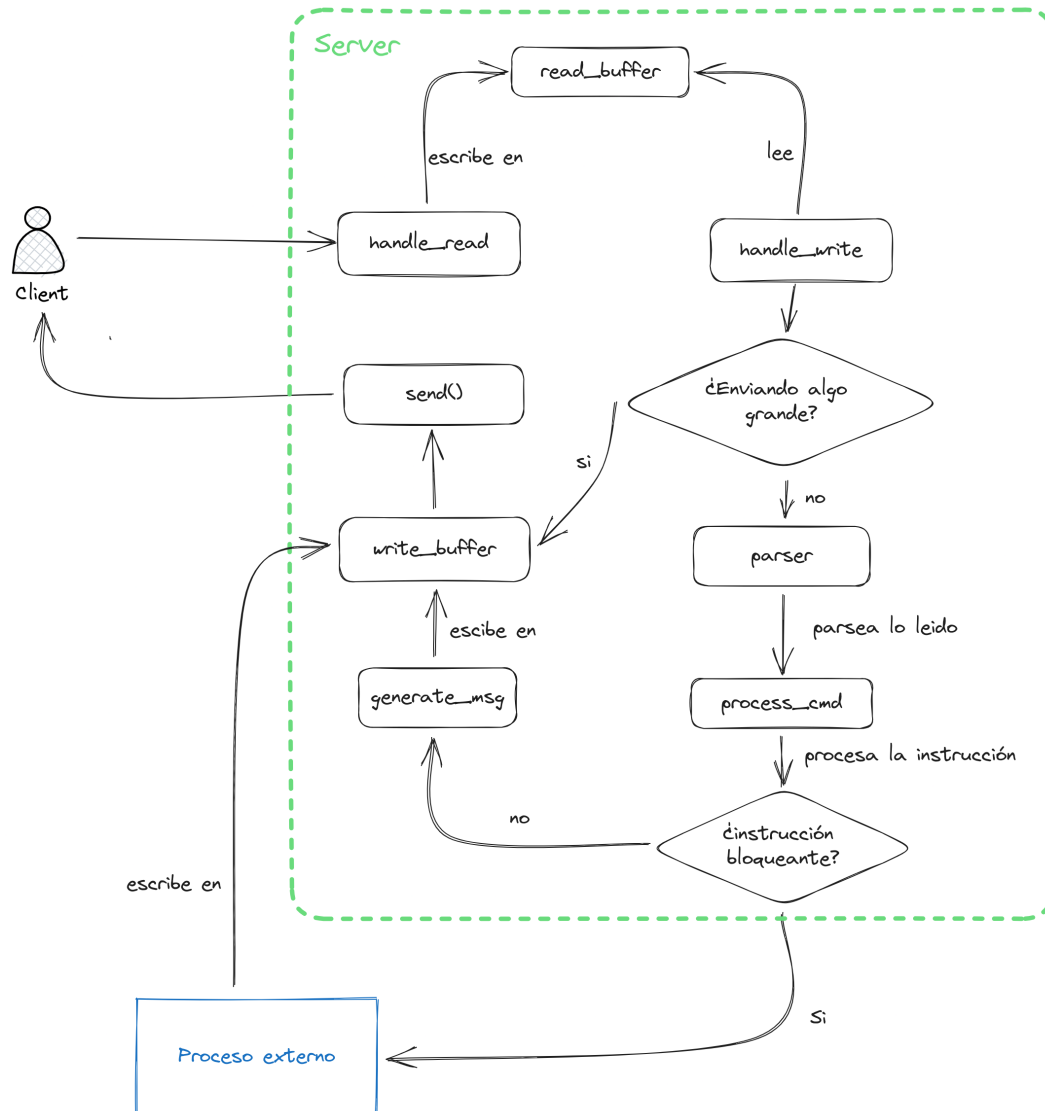


Figura 1: Diagrama representando el flujo de un cliente

2.4. Protocolo de monitoreo

Para el diseño del protocolo de monitoreo tuvimos en cuenta, no sólo las funcionalidades pedidas, sino aquellas funcionalidades que sean útiles en la administración del servidor. Debido a la experiencia ganada en desarrollar POP3, algunas de la

características del protocolo son similares a las definidas en POP3.

El protocolo de monitoreo es un protocolo basado en texto, con palabras separadas por exactamente un espacio. Requiere de autenticación, ya que da la posibilidad de realizar cambios en el servidor. Es responsabilidad de quién crea el servidor la definición de usuarios para el protocolo de monitoreo. En nuestro servidor se ha definido un sólo usuario administrador cuyo usuario y contraseña son los mismos: “admin”, como guiño a los viejos routers.

En cuanto a las funcionalidades sobre los datos históricos del servidor podemos consultar tres datos: Número de conexiones simultáneas actuales, número de conexiones totales a lo largo del tiempo y cantidad total de bytes enviados. Por otro lado, las funcionalidades de administración son: Obtener una lista de todos los usuarios, obtener el estado de un usuario, remover un usuario junto a si maildir, agregar un usuario nuevo y por último popular el maildir de un usuarios con 10 archivos de tamaño variante entre 16KB y 64MB. Cabe destacar que no puede hacerse una modificación a un usuario si este no está desconectado del servidor.

Para una descripción mas técnica y detallada del protocolo, puede consultarse el [documento técnico](#)

Para la implementación del protocolo utilizamos un acercamiento similar al usado en POP3, con la principal diferencia de que omitimos el uso de una máquina de estados, pues no fue necesario definir distintos comportamientos para el protocolo dependiendo de alguna variante. Solamente añadimos una condición que pide que el usuario se haya autenticado para los comandos en los que sea necesario.

3. Pruebas de los protocolos

3.1. Integridad de los datos

Para validar que el servidor POP3 envía exactamente el mismo correo que tiene almacenado en el Maildir, [realizamos un script](#) que, en un nuevo usuario creado a través del cliente del monitor, crea un archivo de 2 GiB (con caracteres ASCII imprimibles obtenidos de `/dev/urandom`) en el Maildir de este usuario, y lo recibe

utilizando [curl](#).

Ambos archivos son, a continuación, comparados mediante un hash SHA-256 con `sha256sum` [3] y el resultado final es mostrado en pantalla al usuario, como se aprecia en la Figura 2.

```
[itba@rhino] { main ♦ } POP3
$ ./test/test_integrity.sh
Creating user in POP3 server...
0w0 Successfully logged

0w0 User INTEGRITY1 added

Creating 2 GiB email... OK
Fetching file with curl...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 2048M    0 2048M    0     0   108M      0  --:--:--  0:00:18 --:--:-- 120M
OK
Expected:      5c0d4b3c41f87ec32687ca360b11671df9ad63c48308f2e3805c5069ce0a07e8
Received:      5c0d4b3c41f87ec32687ca360b11671df9ad63c48308f2e3805c5069ce0a07e8
We received the exact same file!
[itba@rhino] { main ♦ } POP3
$ _
```

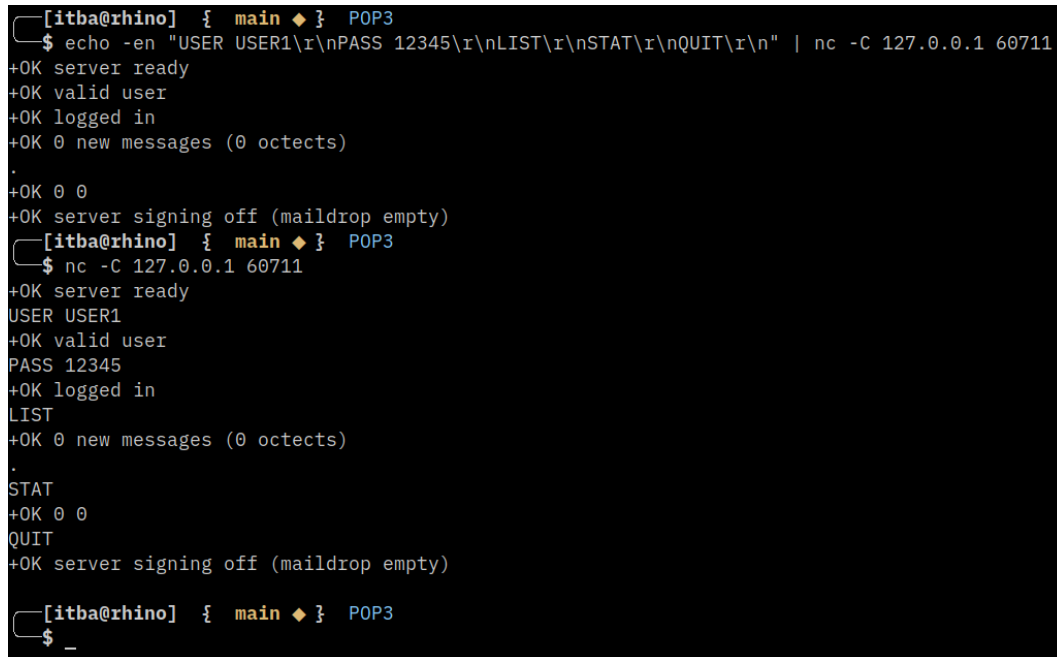
Figura 2: Captura de pantalla de la prueba de integridad luego de su ejecución.

3.2. Pipelining

El más simple de todos los tests, también [escrito en un script](#), envía al servidor una serie de comandos POP3, en el orden que espera que este los ejecute, y obtiene la respuesta del servidor POP3. En la Figura 3 se puede apreciar que la respuesta es la misma tanto si los comandos son ejecutados a mano como mediante el pipelining.

3.3. Conexiones simultáneas

Por último, desarrollamos dos scripts para probar realizar 500 conexiones simultáneas al servidor. [El primero de ellos](#) se conecta al protocolo de monitoreo para indicarle que debe crear 500 distintos usuarios y llenar sus Maildirs con archivos



```
[itba@rhino] { main } POP3
$ echo -en "USER USER1\r\nPASS 12345\r\nLIST\r\nSTAT\r\nQUIT\r\n" | nc -C 127.0.0.1 60711
+OK server ready
+OK valid user
+OK logged in
+OK 0 new messages (0 octects)
.
+OK 0 0
+OK server signing off (maildrop empty)
[itba@rhino] { main } POP3
$ nc -C 127.0.0.1 60711
+OK server ready
USER USER1
+OK valid user
PASS 12345
+OK logged in
LIST
+OK 0 new messages (0 octects)
.
STAT
+OK 0 0
QUIT
+OK server signing off (maildrop empty)
[itba@rhino] { main } POP3
$ _
```

Figura 3: Captura de pantalla del pipelining y su equivalente en el servidor.

aleatorios, utilizando el comando `POPULATE_USER`, para luego conectarse a través de pipelining al servidor y ejecutar, en orden, los comandos `STAT`, `LIST` y `RETR 1`.

Para asegurarnos que esto se realizaba en procesos distintos, escribimos una función en Bash que es llamada en segundo plano¹.

El otro script también se conecta al monitor para crear los usuarios, pero al servidor POP3 se conecta utilizando `curl` y no ejecuta comandos adicionales, además de los relacionados con `AUTH`, en el servidor.

¹Puntualmente, `connect_to_ip &`.

4. Problemas encontrados

Durante el desarrollo del proyecto nos encontramos con algunas dificultades de diseño e implementación, e incluso algunas diferencias en lo referido a la calidad del código que detallamos a continuación.

4.1. Problemas menores

4.1.1. Makefile

A medida que avanzamos en el desarrollo del proyecto, tuvimos que modificar nuestro sistema de compilación para que se adapte al mismo. En particular, algunas de estas reescrituras mantenían los archivos objeto generados por el compilador en la carpeta `obj` e impedían a uno de los miembros del equipo recompilar el proyecto.

4.1.2. Estilo de código

Se puede notar la diferencia en la calidad del código si se hace un seguimiento temporal del proyecto. En un principio nos encontrábamos explorando las distintas maneras de llegar a la solución y no teníamos una visión tan clara sobre los pasos a seguir. Mientras más avanzamos con el proyecto generamos un entendimiento muy profundo de la solución el cuál nos permitió tener una calidad de código mayor en los módulos desarrollados sobre el final. Si bien una refactorización del código puede ser beneficiosa para la legibilidad, teniendo en cuenta la fecha límite y el resto de responsabilidades de los autores, decidimos seguir con la filosofía de *“if it ain’t broke, don’t fix it”* para poder entregar un resultado funcional.

4.2. Envío del caracter terminador

Según el RFC 1939 [1], al finalizar el envío de un comando cuya respuesta es multilínea (como `LIST` o `RETR`) se debe enviar un caracter terminador para avisar al cliente que se ha enviado la totalidad del mensaje.

Esto representó un desafío que nos obligó a comprender con mayor profundidad el flujo, en código, de la máquina de estados sobre la cual implementamos el servidor. Como el envío de información funciona, de manera abstracta, como una máquina de estados dentro del estado **TRANSACTION**, optamos por agregar otro “estado” dentro de la misma que, al terminar de enviar el mensaje, vuelve a entrar en **TRANSACTION** para enviar el caracter terminal.

4.3. Maildir

Encontramos dificultades al momento de comprender cómo se debía implementar Maildir [2] pues el estándar del mismo no define algunos detalles, como cuáles son las *experimental semantics* (que decidimos omitir completamente, pues no son de relevancia para nuestro servidor), o si es obligatorio agregar el flag **T** y mover el archivo a **cur** sabiendo que este debe ser eliminado.

En este último caso, optamos por, efectivamente, realizar el cambio de nombre del archivo y moverlo de **new** a **cur**, y luego utilizar la función estándar **remove** [4].

La motivación detrás de esta decisión es que, por tratarse de archivos, utilizar la función estándar **rename** [5] nos garantiza que el archivo cambiará su nombre ², mientras que la función **remove** mantiene el archivo en su lugar hasta que el último proceso que lo tiene abierto lo cierre [4]. Utilizar **remove** directamente podría derivar en el caso en que otro proceso tiene un correo electrónico abierto, el usuario lo elimina, cierra su sesión, y antes de que el proceso ajeno cierre el archivo, el usuario se vuelve a conectar con el servidor, pudiendo ver nuevamente el correo que creía eliminado.

4.4. Envío de un caracter extraño

Durante las pruebas, notamos que en ciertas ocasiones se enviaba uno o más caracteres extraños (que no formaban parte de ningún mensaje o correo) al cliente.

Por ejemplo, en lugar de recibir

²Los casos de error descriptos en el manual aplican a directorios (**EBUSY**); ocurren por fuera de nuestro control (**ENOMEM**); o los descartamos por ser el servidor el creador del Maildir (y, por ende, tiene permiso de lectura y escritura, **EACCESS**)

“+OK server signing off (22 unread, 0 removed, 22 total)”

el cliente recibía

“Lc+OK server signing off (22 unread, 0 removed, 22 total)”

Sospechando de estar pisando memoria o enviando bytes extras al utilizar `send` [6], buscamos y analizamos todos los posibles caminos que terminaban en un llamado a la función mencionada, sin éxito. El problema resultó ser en la no-inicialización del arreglo que guarda los mensajes de respuestas con el resulta del comando ingresado por el cliente: esta en algunas ocasiones se llenaba con datos basura y cuando el cliente realizaba un `QUIT` el mismo imprimía los datos basura del arreglo. Todos los demás comandos evitaban este error pues siempre reemplazaban el contenido del arreglo al finalizar. (`QUIT` utiliza otro arreglo, que se encuentra dentro del estado `UPDATE`).

4.5. Lectura de archivos grandes

Como ya sabemos, al utilizar `RETR` podemos pedir un archivo que sea de un tamaño mucho mayor al buffer del socket, o incluso al buffer de salida de nuestro cliente. En estos casos, conviene ir leyendo el archivo de a partes e ir enviando dichas partes en distintos paquetes. El problema es que leer un archivo de tamaño grande puede bloquear al servidor. Por lo que la solución pensada fue de seguir un patrón *Producer - Consumer* [7]. Lo que implementamos fue un módulo encargado de leer un archivo, el cuál abre un nuevo *file descriptor* que se registraba junto con los *sockets* en el selector. Luego se desuscribe al cliente de lectura y de escritura, para que no hiciese nada mientras no haya datos para leer. Por ultimo se suscribe en lectura al módulo lector de archivos para que leyera una parte del archivo y llenara lo leído en el buffer de salida del cliente. Una vez que sucedía esto, se vuelve a suscribir al cliente para escritura y una vez que el cliente termina de enviar todo lo que hay en el buffer, se vuelve a desuscribir y a suscribir al lector del archivo para lectura.

4.6. Uso de programa externo para lectura de archivo

Otro problema fue el de uso de un programa externo. La principal idea es utilizar *cat*, pero podría utilizarse cualquier programa para analizar un mail y devolverlo procesado por salida estándar. Para esto, tras investigar, nos topamos con las funciones *popen* y *fileno*. La primera recibe como parámetro un string que contiene el comando a ejecutar, y automáticamente realiza todas las operaciones necesarias como **fork** o **dup2** para ejecutar dicho comando. El problema es que esta función no nos devuelve un *file descriptor* sino un puntero al tipo de dato **FILE**, por lo que nuestra solución anterior para lectura de archivos se volvía obsoleta. En este lugar es donde entra la función **fileno**, la cuál recibe como parámetro un puntero a **FILE** y devuelve el *file descriptor* abierto por ese puntero. De esta forma se resuelve el problema y se puede seguir utilizando la solución a la lectura de archivos grandes.

4.7. Población de maildirs

Un problema similar al de lectura se presentaba con la funcionalidad de poblar el maildir de un usuario. El generar archivos de un tamaño considerable presentaba una ejecución bloqueante. Se utilizó un acercamiento similar al uso de programa externo para la lectura de un archivo. Se realizó un comando que crea archivos utilizando el comando *dd*. De esta manera, utilizando las funciones **popen** y **fileno** podemos ejecutar la tarea haciendo que el cliente de monitoreo espere una respuesta, pero sin bloquear a ningún cliente que esté utilizando el servidor POP3.

5. Limitaciones de la aplicación

Debido a la implementación de multiplexing utilizando `select` [8], el número máximo de conexiones que puede soportar actualmente el servidor es de 1021³ ⁴.

También, debido a que cada usuario puede tener abierto un *file descriptor* si está leyendo un archivo, es notado que en el peor de los casos, el servidor puede tener 510 conexiones autenticadas al mismo tiempo, ya que cada usuario estaría usando dos *file descriptors* abarcando así el límite establecido.

Por otra parte, el cliente fue desarrollado sin ninguna clase de interfaz interactiva, teniendo como única forma de recibir los comandos de monitoreo los argumentos con que se ejecuta el mismo.

Se tomó esta decisión pues nos ahorra tiempo y facilitaba tanto el desarrollo y las pruebas del mismo, como las del servidor y monitor. Al tener la posibilidad de cargar todos los comandos en la terminal, la misma se encarga de recordar qué escribimos anteriormente (brindándonos repetitividad al momento de buscar un error, como el mencionado en la sección 4.4), y es posible realizar pruebas de todos los comandos del servidor y del monitor en una misma ejecución del cliente.

Por último, las credenciales y usuarios del servidor son volátiles (pero no sus Maildirs), por lo que reiniciar el servidor elimina de la memoria toda credencial que no sea el usuario por defecto. Esto nos permitió ignorar el manejo de guardar credenciales en disco y nos brindó gran facilidad para realizar pruebas del servidor.

Es también importante notar que este servidor está pensado para funcionar en sistemas operativos basados en el Kernel de Linux que sean POSIX Compliant.

En la sección 6 discutimos más aspectos de los puntos comentados anteriormente.

³Este límite incluye las conexiones de usuarios del servidor POP3 y usuarios del monitor.

⁴Si bien `select` [8] tiene un límite de 1024 *file descriptors*, se deben descontar tres para los sockets pasivos del servidor y el monitor.

6. Posibles extensiones

La aplicación puede ser extendida agregando comandos extra al protocolo de monitoreo, entre los que se podría incluir la posibilidad de revocar las credenciales de un usuario del servidor (sin eliminarlo), más usuarios de monitoreo con algún sistema de permisos para mayor granularidad, etc.

También se puede eliminar la limitación de 1022 conexiones simultáneas utilizando `epoll` [9] si el servidor detecta que está siendo ejecutado en Linux, o `poll` [10] en otro sistema Unix. Esto también requeriría de una revisión del código en general, especialmente de las máquinas de estado implementadas en cada etapa, pues su calidad de código y simplicidad fueron aumentando a medida que avanzamos con el proyecto⁵.

En última instancia, como se mencionó anteriormente, las credenciales de los usuarios viven en memoria volátil, lo cual es inaceptable para cualquier servidor utilizado en producción. Esta es una importante extensión que debemos realizar a la aplicación.

⁵Se puede comparar los archivos `auth.c`, `transaction.c`, `update.c`, `monitor.c`, en ese orden, para observar la evolución de nuestra comprensión teórica y familiaridad con el diseño de la aplicación.

7. Conclusiones

Este trabajo supuso una colección de retos difíciles. Nos enseñó lo complejo de la lógica de manejar un servidor y las cosas que hay que tener en cuenta a la hora de hacerlo. El manejo de tareas bloqueantes y cómo proporcionar la mejor performance para un servidor no son tareas sencillas y gracias a estos desafíos hemos aprendido una manera de poder resolverlo. También sentó las bases de cómo comenzar a definir un protocolo, en caso de ser útil y necesario en un futuro.

Hoy en día existen cientos de herramientas para poder poner en marcha servidores de todo tipo en cuestión de minutos, abstrayendo al desarrollador de todo lo que está sucediendo detrás de escena. Este desarrollo no pretende suplantarse dichas herramientas pero sí saber apreciarlas mejor y entender cómo funcionan.

Referencias

- [1] D. M. T. Rose and J. G. Myers, “Post Office Protocol - Version 3,” RFC 1939, May 1996. [Online]. Available: <https://www.rfc-editor.org/info/rfc1939>
- [2] “Using maildir format,” <https://cr.yp.to/proto/maildir.html>, accessed: 2023-06-22.
- [3] *SHA256SUM(1) sha256sum - compute and check SHA256 message digest*, GNU coreutils 9.3, April 2023, <https://linux.die.net/man/1/sha256sum>.
- [4] *remove(3) remove - remove a file or directory*, Linux man-pages, March 2023, <https://linux.die.net/man/3/remove>.
- [5] *rename(2) rename, renameat, renameat2 - change the name or location of a file*, Linux man-pages, March 2023, <https://linux.die.net/man/2/rename>.
- [6] *send(2) send, sendto, sendmsg - send a message on a socket*, Linux man-pages, March 2023, <https://linux.die.net/man/2/send>.
- [7] “Producer-consumer problem,” https://en.wikipedia.org/wiki/Producer-consumer_problem, accessed: 2023-06-23.
- [8] *select(2) select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO, fd_set - synchronous I/O multiplexing*, Linux man-pages, March 2023, <https://linux.die.net/man/2/select>.
- [9] *epoll(7) epoll - I/O event notification facility*, Linux man-pages, March 2023, <https://linux.die.net/man/7/epoll>.
- [10] *poll(2) poll, ppoll - wait for some event on a file descriptor*, Linux man-pages, March 2023, <https://linux.die.net/man/2/poll>.
- [11] P. Resnick, “Internet Message Format,” RFC 5322, Oct. 2008. [Online]. Available: <https://www.rfc-editor.org/info/rfc5322>
- [12] “curl,” <https://curl.se/>, accessed: 2023-06-23.