

Instituto Tecnológico de Buenos Aires



## **72.07 Protocolos de Comunicación TPE**

### **Grupo 11**

(62028) Nicolás Matías Margenat - [nmargenat@itba.edu.ar](mailto:nmargenat@itba.edu.ar)

(62094) Juan Burda - [jburda@itba.edu.ar](mailto:jburda@itba.edu.ar)

(62493) Saul Ariel Castañeda - [scastaneda@itba.edu.ar](mailto:scastaneda@itba.edu.ar)

(62504) Elian Paredes - [eparedes@itba.edu.ar](mailto:eparedes@itba.edu.ar)

# Índice

---

<b>1. Descripción detallada de los protocolos y aplicaciones desarrolladas</b>	<b>2</b>
1.1. Server POP3	2
1.2. Popcorn Protocol (versión 1)	3
1.2.1. Introducción	4
1.2.2. Estructura de los datagramas	4
1.2.3. Códigos de status	5
1.2.4. Comandos soportados	6
1.2.5. Ejemplos de uso de los comandos	7
1.3. Aplicación cliente	11
<b>2. Problemas encontrados durante el diseño y la implementación</b>	<b>11</b>
<b>3. Limitaciones de la aplicación</b>	<b>12</b>
<b>4. Posibles extensiones</b>	<b>12</b>
<b>5. Conclusiones</b>	<b>13</b>
<b>6. Ejemplos de prueba</b>	<b>13</b>
<b>7. Guía de instalación</b>	<b>15</b>
<b>8. Instrucciones para la configuración</b>	<b>16</b>
<b>9. Ejemplos de configuración y monitoreo</b>	<b>16</b>
<b>10. Documento de diseño del proyecto</b>	<b>18</b>
<b>11. Referencias</b>	<b>20</b>

# 1. Descripción detallada de los protocolos y aplicaciones desarrolladas

---

## 1.1. Server POP3

El servidor POP3 implementado comienza por extraer los argumentos que se pasaron por parámetros al programa. Estos argumentos permiten definir los usuarios y las contraseñas presentes en el directorio también pasado por argumentos, este directorio es el que contendrá los directorios de los distintos usuarios. Es importante notar que esta estructura que contiene los directorios de los usuarios del servidor debe haber sido creada previamente y debe coincidir uno a uno con los usuarios y contraseñas recibidos por argumentos, de lo contrario esta etapa fallará y se abortará la ejecución.

Pasada la etapa de inicialización, se suscribe al servidor para lectura en el selector y se agrega un handler que se encarga de aceptar de manera pasiva nuevos clientes. Se crea además un handler para cerrar el servidor, en caso de que sea necesario abortar y liberar todos los recursos del sistema que fueron reservados por el servidor.

En el momento en que llegan nuevos clientes, se crea una nueva sesión para cada uno de ellos. Este objeto representa una conexión con el cliente y es liberado una vez que el cliente cierra su conexión con el servidor.

Al inicializar una nueva sesión, se pusha un estado base READ, que no se retira en ningún momento, y un estado PROCESS. Además, se suscribe a la sesión a escritura y se setea el handler de escritura. Luego, cuando el servidor trata de escribir al cliente se da cuenta que tiene PROCESS al tope de la pila, por lo que ejecuta *session\_process*. En esta función, se procesa el greeting, y cuando termina de hacerlo lo deja en el buffer de escritura, poppea la pila (es decir, se poppea PROCESS) y pusha WRITE. De esta manera, la próxima vez que el selector diga que es su turno le tocará escribir y podrá hacerlo.

Cuando toque escribir vamos a tener dos casos. En el caso de que se haya podido mandar todo se poppea el estado de WRITE y se lo suscribe a lectura, seteando el handler de manera acorde. Si por el contrario no se puede escribir todo al cliente, entonces se pushearán un estado WRITING, que le indica a la sesión que tiene cosas en el buffer de escritura que debe terminar de mandar. Consecuentemente, la próxima vez que sea su turno sabrá dónde estaba. Cuando termine de mandar todo se poppea WRITE y WRITING, y se lo deja suscrito a escritura, seteando su handler.

Terminado el greeting, cuando llegue un comando del cliente se va despertar a la sesión, se va a leer y guardar lo que mandó el cliente, y luego se pasará este buffer al parser. El parser no va a consumir todo, sino que va a consumir solo hasta que encuentre un comando. En el caso de que no encuentre una línea (terminada por `\r\n`) se guardará hasta dónde leyó y retornará. Esto se repetirá hasta que sea capaz de encontrar el fin de línea. Notemos que en este caso seguiremos teniendo READ en el tope de la pila.

Cuando el parser lee exactamente un comando entonces puede pasar que no haya nada en el buffer de lectura, en esta caso se lo guardará en la sesión del cliente y se pushearán el estado de PROCESS. Luego se llamará a dicho estado, el cual ejecutará la función *dispatch*. Esta función se fija cuál es el estado del cliente en la *state\_machine* (los estados del cliente

son los descriptos en el RFC 1939 [RFC1939]), y verifica que el comando que haya mandado pertenezca a dicho estado. En el caso de que el comando sea incorrecto, entonces se mandará un mensaje de error. Por otro lado, si el mensaje es correcto se pasa a procesarlo. Al momento de terminar el procesamiento, se va a pushear WRITE suscribir a escritura en el selector y retorna.

Puede haber pasado también que el parser haya agarrado una línea y un poco más, en cuyo caso no queremos que vuelva a leer del cliente inmediatamente después de procesar el comando. Por lo tanto, se pushea a la pila un estado de READING, que avisa que todavía se tiene que terminar de leer lo que hay en el buffer antes de poder volver a leer del cliente, y se vuelve a hacer lo mismo que antes, pero nótese que ahora en el stack se tiene READ, READING y PROCESS. El proceso es el mismo que en el párrafo anterior: cuando se termine el procesamiento se poppea PROCESS y se pushea WRITE para poder escribir la respuesta al cliente. De nuevo, si no se logra mandar todo de una pasada se pushearán WRITING a la pila.

El problema cuando se poppea WRITE/WRITING y no se había terminado de leer el buffer de lectura (es decir, había más de un comando/línea), pues en este caso no es posible suscribirse a READ ya que se bloqueará el proceso. Es en este punto que es útil el estado de READING. Lo que sucederá es lo siguiente, en la siguiente iteración, como todavía teníamos al cliente suscrito a escritura en el selector, se llamará a ese handler, pero como en el tope de la pila se tiene READING, iremos a leer. Es decir, iremos al handler de READ sin estar suscriptos a lectura en el selector. Entonces, se vuelven a tener dos casos: si al leer de nuevo del buffer de lectura no se consigue completar otro comando se poppeará READING y quedará READ al tope de la pila, pues en este caso estamos esperando a que el usuario termine de mandar lo que falta para completar el comando. Si se llegase a completar el comando se sigue normalmente.

Finalmente, se tiene un último caso especial que resulta relevante mencionar. Todo lo anterior vale para cualquier comando menos RETR y LIST, que son comandos multilínea. En estos comandos, si no se termina de procesar la información en una iteración entonces se pushea un estado PROCESSING a la pila, y luego WRITE. Se llamará al handler de escritura, y si se termina de escribir se poppea WRITE o se pushea WRITING en el caso de que no se termine de escribir. Nótese que el cliente sigue suscrito a escritura en el selector. Entonces cuando vuelva a entrar al handler de escritura, se dará cuenta que en el tope de la pila tiene PROCESSING, por lo que en vez de escribir se irá a procesar y llamará a *session\_process*. Consecuentemente, se procesará la información, se mandará al cliente y se repetirá el proceso hasta que ya no haya nada más por procesar.

Este proceso se encuentra diagramado en la [sección 10](#), por lo que se sugiere dirigirse allí para comprender mejor el proceso.

## 1.2. Popcorn Protocol (versión 1)

Las palabras clave "DEBE", "NO DEBE", "OBLIGATORIO", "DEBERÁ", "NO DEBERÁ", "DEBERÍA", "NO DEBERÍA", "RECOMENDADO", "PUEDE" y "OPCIONAL" en esta sección serán interpretadas como se describe en el RFC 2119 [RFC2119].

### 1.2.1. Introducción

El objetivo de Popcorn Protocol es obtener métricas relevantes y realizar modificaciones de configuración en tiempo de ejecución.

El protocolo utiliza UDP como protocolo de transporte. Por esta razón, y para evitar que cualquier persona realice modificaciones sobre el servidor, el usuario DEBE estar autenticado mediante usuario y contraseña propios del servidor. En cada datagrama, este campo DEBE ser enviado para legitimar la petición del usuario.

A continuación se detallarán los campos que se DEBEN enviar en todos los datagramas de request del lado del cliente, y los de response del lado del servidor.

### 1.2.2. Estructura de los datagramas

El datagrama de request del cliente DEBE tener la siguiente estructura:

```
popcorn\r\n
version: <version>\r\n
auth: <user>:<pass>\r\n
req-id: <token>\r\n
command: <command> [args]\r\n
```

donde:

- La primera línea DEBE ser el nombre del protocolo (popcorn). Este nombre DEBE estar en minúscula.
- Se aspira que el protocolo sea utilizado en el futuro, por lo que entre los headers DEBE incluirse el header de versión. De esta manera, los servidores que lo implementen serán capaces de rápidamente identificar si los comandos que recibe de un cliente los soporta o no.
- Como se dijo en la introducción, el usuario DEBE incluir en todos los datagramas el header de *auth* (autenticación) para que el servidor sea capaz de discernir entre peticiones legítimas y aquellas que no lo son.
- El cliente DEBE encargarse de mandar en su petición el header *req-id*, donde *token* es un valor numérico, que sirve para diferenciar distintas peticiones. Más adelante se proveerán ejemplos para ayudar a entender este punto (ver [sección 1.2.5](#)).
- El cliente DEBE mandar en su datagrama el comando. Dicho comando PUEDE tener argumentos. Qué comandos existen, cuáles son los argumentos que reciben, y los posibles valores de respuesta se explicarán en detalle en las próximas secciones.

El datagrama de respuesta del servidor DEBE tener la siguiente estructura:

```
popcorn\r\n
version: <version>\r\n
req-id: <token>\r\n
status: <status-code>\r\n
```

```
[value: <value>]\r\n
```

- Los headers de popcorn, version y req-id ya se mencionaron al hablar de el datagrama de request. Estos headers DEBEN ser enviados por el servidor.
- El header de status DEBE ser enviado en cualquier respuesta. El header es una manera rápida de ver si la respuesta fue satisfactoria, y en caso de haber error saber de qué tipo de error se trata. En una sección posterior se explicarán los distintos códigos de status.
- El header de value PUEDE ser enviado por el servidor. La decisión de enviarlo o no depende estrictamente del comando al que se esté respondiendo.

Nótese que cada línea del datagrama DEBE terminar con “\r\n”.

### 1.2.3. Códigos de status

Los códigos de status que maneja el servidor son:

Código	Significado	Descripción
20	OK	La operación salió bien
40	CLIENT ERROR	Hubo un problema del lado del cliente. Esto puede significar que el cliente mandó un argumento inválido
41	BAD CREDENTIALS	No se envió ningún usuario en el header de “auth”, o el usuario enviado no existe o no tiene los permisos necesarios para ejecutar los comandos
42	USER NOT EXISTS	Los comandos password/delete fallaron pues el usuario que se envió en sus argumentos no exists
43	USER LOGGED IN	El comando password falló porque se intentó cambiar la contraseña de un usuario que está logueado
49	VERSION NOT SUPPORTED	Si se envía el header de “version” con una versión que el servidor no soporta

50	SERVER ERROR	Ocurrió un error inesperado en el servidor
----	--------------	--

#### 1.2.4. Comandos soportados

Los comandos que se envíen dentro del header “command” DEBEN estar en minúscula. En cualquier caso, el servidor DEBE enviar el header de “status” con el código pertinente.

Al momento de escribir este informe se cuentan con los siguientes comandos:

- Bytes: Indica la cantidad de bytes transferidos en el servidor. Se cuentan todos los bytes transferidos, incluso los de la escritura de las respuestas. Este comando NO DEBE recibir argumentos. El *value* retornado por el servidor DEBE ser un *unsigned int*.

Los códigos de status que PUEDE enviar el servidor como respuesta a este comando son: 20, 41, 49 y 50.

- History: Indica la cantidad de conexiones históricas del servidor. Este comando NO DEBE recibir argumentos. El *value* retornado por el servidor DEBE ser un *unsigned int*.

Los códigos de status que PUEDE enviar el servidor respuesta a este comando son: 20, 41, 49 y 50.

- Current: Indica la cantidad de usuarios conectados al servidor al momento de recibir el request. Este comando NO DEBE recibir argumentos. El *value* retornado por el servidor DEBE ser un *unsigned int*.

Los códigos de status que PUEDE enviar el servidor respuesta a este comando son: 20, 41, 49 y 50.

- Password: Realiza el cambio de contraseña para un usuario del servidor. Este comando DEBE recibir 2 argumentos, primero DEBE recibir el nombre de usuario del servidor y el segundo argumento DEBE ser la nueva contraseña. El servidor NO DEBE incluir el header “value” en su respuesta.

Los códigos de status que PUEDE enviar el servidor respuesta a este comando son: 20, 41, 42, 43, 49 y 50.

- Delete: Borra un usuario del servidor. Este comando DEBE recibir un argumento que DEBE ser el nombre del usuario a borrar. El servidor NO DEBE incluir el header “value” en su respuesta.

Los códigos de status que PUEDE enviar el servidor son: 20, 41, 42, 49 y 50.

- Conc: Permite modificar la cantidad máxima de usuarios concurrentes. Este comando DEBE recibir un argumento que DEBE ser un valor numérico de tipo *int* y DEBE ser menor a 1000. El servidor NO DEBE incluir el header “value” en la respuesta.

Los códigos de status que PUEDE enviar el servidor respuesta a este comando son: 20, 41, 49 y 50.

A modo de resumen se provee esta tabla con los comandos y sus descripciones:

Comando	Argumentos	Descripción	Status codes
bytes	NO	Indica la cantidad de bytes transferidos en el servidor	20, 41, 49, 50
history	NO	Indica la cantidad de conexiones históricas del servidor	20, 41, 49, 50
current	NO	Indica la cantidad de usuarios conectados al servidor al momento de mandar la request	20, 41, 49, 50
password	2	Cambia la contraseña de un usuario del servidor	20, 41, 42, 43, 49, 50
delete	1	Borra un usuario del servidor	20, 40, 41, 42, 49, 50
conc	1	Permite modificar la cantidad máxima de usuarios concurrentes	20, 40, 41, 49, 50

### 1.2.5. Ejemplos de uso de los comandos

A continuación se muestran ejemplos de uso de los comandos. Para ello se asume que el usuario con username “admin” y contraseña “pass” se encuentra registrado y tiene los permisos necesarios para ejecutar los comandos.

A menos que se indique lo contrario, la caja de arriba es la request del cliente y la de abajo es la respuesta del servidor.

#### - Bytes

EJEMPLO 1: El usuario “admin” pide la cantidad de bytes transferidos del servidor.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 1234\r\n
command: bytes\r\n
```

```
popcorn\r\n
version: 1\r\n
```



```
req-id: 1234\r\n
status: 20\r\n
value: 70218\r\n
```

- History:

EJEMPLO 2: El usuario “admin” pide la cantidad de conexiones históricas del servidor.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 1982384\r\n
command: history\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 1982384\r\n
status: 20\r\n
value: 210\r\n
```

- Current:

EJEMPLO 3: El usuario “admin” pide la cantidad de usuarios que están actualmente conectados al servidor.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 32\r\n
command: current\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 32\r\n
status: 20\r\n
value: 6\r\n
```

- Password:

EJEMPLO 4: El usuario “admin” cambia la contraseña del usuario “doc”.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 120\r\n
command: password doc brown\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 120\r\n
status: 20\r\n
```

EJEMPLO 5: El usuario “nothacker” intenta cambiar la contraseña de “admin” pero no tiene privilegios.

```
popcorn\r\n
version: 1\r\n
auth: nothacker:irobot\r\n
req-id: 837\r\n
command: password admin pwnd\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 837\r\n
status: 41\r\n
```

EJEMPLO 6: El usuario “admin” intenta cambiar la contraseña de “tweedledee” pero se olvida un argumento.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 8474\r\n
command: password tweedledee\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 8474\r\n
status: 40\r\n
```

- Delete:

EJEMPLO 7: El usuario “admin” borra al usuario “nothacker”.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 94763\r\n
command: delete nothacker\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 94763\r\n
status: 20\r\n
```

EJEMPLO 8: El usuario “admin” intenta borrar al usuario “stallone” (no es un usuario que existe en el servidor).

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 8464\r\n
command: delete stallone\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 8464\r\n
status: 42\r\n
```

- Conc:

EJEMPLO 9: El usuario “admin” setea la cantidad de máximas conexiones concurrentes a 700.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 123\r\n
command: conc 700\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 123\r\n
status: 20\r\n
```

EJEMPLO 9: El usuario “admin” setea la cantidad de máximas conexiones concurrentes a un valor que no tiene sentido.

```
popcorn\r\n
version: 1\r\n
auth: admin:pass\r\n
req-id: 23456\r\n
command: conc jd3fsa\r\n
```

```
popcorn\r\n
version: 1\r\n
req-id: 23456\r\n
status: 40\r\n
```

## 1.3. Aplicación cliente

El cliente utiliza UDP para mandar datagramas al servidor. El proceso que sigue es extremadamente simple: se corre el programa con los argumentos, se manda la request, se recibe la respuesta y se cierra el programa.

En cuanto al código, en *request.c* se construye el string que se va a enviar al servidor en el datagrama. En un primer lugar se construye el struct, y luego el string. Si bien este struct no es utilizado muchas veces en el código, el equipo creyó que de esta manera aumentaba la legibilidad y mantenibilidad del código. En segundo y último lugar, se utiliza un parser cuando llega la respuesta del servidor, que toma el string y rellena un struct con los datos. Finalmente, se interpretan los códigos de estado de la respuesta para otorgarle al cliente una respuesta adecuada.

## 2. Problemas encontrados durante el diseño y la implementación

---

En esta sección se detallarán los problemas encontrados en un orden cronológico. Se incluirán algunas de las discusiones que se tuvieron internamente, las decisiones tomadas y sus razones.

El primer desafío encontrado fue comprender el funcionamiento de los parches provistos por la cátedra, especialmente el del parser. Luego de que se entendió esto, se decidió utilizar el parser para tokenizar los comandos enviados por el cliente, y luego compararlos utilizando *strcmp* con los comandos que establece el RFC de POP3 [RFC1939]. Si bien esto tiene sus desventajas, pues es más lento el parseo de los comandos ya que se requieren dos pasadas por los strings para poder decidir si son o no válidos, se considera que esto no supone un problema de eficiencia que pueda resultar significativo. La razón por la que se hizo de esta manera es que el grafo de transiciones crecía desproporcionadamente por cada comando y argumento que se agregaba. En un futuro, si se desease extender el proyecto, resultaría mucho más difícil tratar de entender lo que se hace que hacerlo con los *strcmp*, y la ganancia de eficiencia no sería justificativa de este salto en complejidad del código.

En un principio, la decisión que tomó el equipo fue seguir la recomendación del profesor de implementar un servidor bloqueante para luego pasarlo a no bloqueante. De esta manera, se entendería mejor el flujo que debía seguir el servidor, y consecuentemente facilitaría la implementación no bloqueante. El problema que se tuvo fue el de realizar la implementación bloqueante es que se hizo con *forks*, por lo que sería imposible realizar los comandos LIST y RETR. Así, al momento de tener que implementar estos comandos se decidió primero migrar el servidor a su forma no bloqueante para luego implementarlos. En retrospectiva, se considera una decisión acertada la de realizar ambas implementaciones, pues el pasaje de no bloqueante a bloqueante no resultó difícil, pues ya se sabía el flujo que se debía manejar.

La implementación de comandos no trajo muchos problemas, exceptuando por el LIST y el RETR. Por un lado, para poder implementar el LIST se tuvo que implementar un stack al que se pushean los estados para poder devolver las respuestas correctamente; esto era necesario ya que, de lo contrario, el servidor podría intentar leer cuando debería seguir procesando y enviando lo procesado. Esto demoró un poco al equipo. Por otro lado, la implementación del RETR tuvo sus complicaciones dado que, en un principio, se estaban considerando también los casos en los que el mail estaba mal formado. Luego de realizar una consulta por mail, se pudo solucionar el problema ya que no era necesario considerar esos casos. Esto le quitó al equipo mucho tiempo y evidenció la complejidad de realizar este tipo de validaciones en los servidores.

En la fase final del proyecto se verificó el correcto funcionamiento de todo lo implementado. Uno de los problemas que se tuvo fue entender por qué el servidor no utilizaba el 100% del CPU al procesar un archivo grande con el comando RETR. Discutiendo, probando e investigando se llegó a la conclusión que esto sucedía porque el procesamiento estaba siendo limitado por las operaciones de escritura a pantalla. Consecuentemente, se hicieron dos pruebas que se muestran en las Imágenes 2.1.a y 2.1.b que muestran que utilizando *netcat* el CPU no llega al 100%, pero si se realiza lo mismo con *curl* y se redirecciona la salida a un archivo sí.

### 3. Limitaciones de la aplicación

---

Dentro de las limitaciones de la aplicación se encuentra que:

- En caso de que RETR no sea capaz de leer todo el mail, se utiliza *lseek* para retomar la posición en la que terminó de leer la vez anterior. Según *man 3 lseek* este comando **puede** no estar disponible en todos los sistemas POSIX.
- Si bien se permite cambiar la cantidad máxima de conexiones concurrentes, esta nunca podrá superar el valor de 1024, pues se encuentra limitado por la implementación de *glibc* del select (ver *man 2 select*).
- El cliente desarrollado no permite mantener una sesión con el servidor.

### 4. Posibles extensiones

---

Entre las posibles extensiones se encuentran:

- Desarrollar un cliente que permita mantener una sesión con el servidor.
- Aceptar una mayor cantidad de comandos de POP3, así como también dar más opciones de configuración al servidor.
- Implementar filtros en los mails, de esta manera se podrían filtrar mensajes que se consideren spam.

## 5. Conclusiones

El trabajo resultó de gran ayuda para consolidar los conocimientos adquiridos durante la cursada. Se pudo trabajar en profundidad con el protocolo POP3, aprendiendo en el camino interpretar los RFCs de manera correcta. Esto fue de gran ayuda para diseñar, implementar y documentar un protocolo propio en una segunda instancia.

En el proceso surgieron numerosas dificultades que pudieron superarse, aunque, sin lugar a dudas, la principal fue la dificultad para entender algunos conceptos. Fue gracias al trabajo en equipo y las discusiones que se mantenían en las altas horas de la noche lo que permitió que el trabajo saliese a flote.

## 6. Ejemplos de prueba

A continuación se enumerarán las pruebas realizadas para demostrar el correcto funcionamiento del servidor POP3.

- Soporte de IPv4 e IPv6

```
saula@LAPTOP-3QGRIRKS:~/protos/popcorn$ netstat -nltp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:1110             0.0.0.0:*               LISTEN      389/popcorn
tcp6       3      0 :::1110                  :::*                     LISTEN      389/popcorn
```

*Imagen 6.1. Soporte IPv4 e IPv6.*

En la Imagen 6.1 se puede ver el servidor escuchando tanto conexiones IPv4 como IPv6 utilizando comando `netstat -nltp`. En las imágenes 6.2.a y 6.2.b que se muestran a continuación, se puede ver que esto efectivamente funciona utilizando el servidor a través del comando `netcat`.

```
saula@LAPTOP-3QGRIRKS:~/protos/popcorn$ netcat -C localhost 1110
+OK READY
```

*Imagen 6.2.a. Conexión IPv4.*

```
saula@LAPTOP-3QGRIRKS:~/protos/popcorn$ netcat -6 -C localhost 1110
+OK READY
```

*Imagen 6.2.b. Conexión IPv6.*

- Obtener un mensaje más chico que el buffer a través de `curl`

Se realizó una prueba ejecutando el comando *curl* desde la terminal, a continuación se mostrarán los pasos que se siguieron para lograr este resultado:

1. Se ejecutaron los siguientes comandos:

```
1. dd if=/dev/urandom of=file bs=4096 count=4
2. base64 file > file2
3. awk '{ printf "%s\r\n", $0 }' file2 > file3 (convierte los \n en \r\n)
4. curl pop3://saul:castaneda@localhost:1110/1 > output (1 es el mensaje file3)
```

2. Luego ejecutando *tail -2* en los archivos (file3 y output) se puede ver lo siguiente:

```
1. tail -2 file3
YwluEsEu5YcJPQ538ffCEtmwoK5+VN3FXs+eWIHEX7BS/bBYanljD4BxYBtjNi66K9CPao++n
iG
Ud4fUMYLOtb2CUhXSVLn2vCS1T8xx3rCGw
2. tail -2 output
Ud4fUMYLOtb2CUhXSVLn2vCS1T8xx3rCGw
```

Como se puede observar, el resultado no es el esperado. Veamos a qué se puede deber esta diferencia.

3. Se ejecuta ahora *od -c* en los dos archivos (file3 y output).

```
0053560 \r \n e 0 S n B X 7 Q W x h A T l
0053600 n u 5 o M / O G x B 8 7 D R E A
0053620 E E + w = = \r \n
0053630
```

Imagen 6.3.a. Output del comando *od -c file3*.

```
0053560 \n e 0 S n B X 7 Q W x h A T l n
0053600 u 5 o M / O G x B 8 7 D R E A E
0053620 E + w = = \r \n \r \n
0053631
```

Imagen 6.3.b. Output del comando *od -c output*.

Como se puede ver en las Imágenes 6.3.a y 6.3.b, al realizar *curl* se está incluyendo un “\r\n” extra. Se atribuye entonces este comportamiento errático a dicho comando. De todos modos, es importante notar que esta prueba se realizó con un archivo pequeño en comparación con el tamaño del buffer, por lo que la prueba siguiente repetirá este proceso pero con un archivo mucho más grande que el buffer con el objetivo de ver si se observan más irregularidades, o se le puede atribuir la irregularidad al comando *curl*.

- Obtener un mensaje más grande que el buffer a través de *curl*

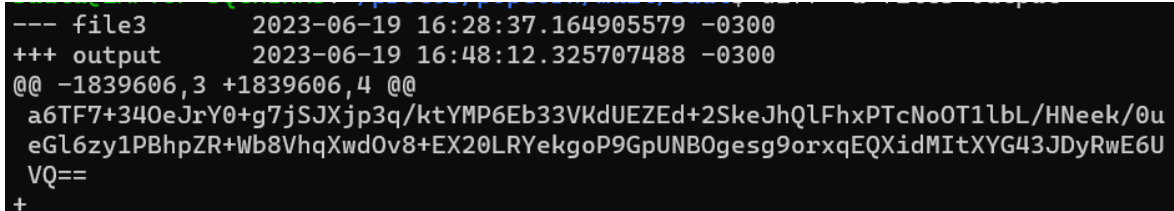
En este caso se ejecutaron los siguientes comandos (nótese que la única diferencia con la prueba anterior se encuentra en la primer línea, pues ahora el archivo que se genera es más grande):

```
1. dd if=/dev/urandom of=file bs=4096 count=25600
2. base64 file > file2
3. awk '{ printf "%s\r\n", $0 }' file2 > file3 (convierte los \n en \r\n)
4. curl pop3://saul:castaneda@localhost:1110/1 > output (1 es el mensaje file3)
```

Ejecutando `ls -la` sobre el directorio que contiene los archivos generados, obteniendo los siguientes valores para `file3` y `output`:

```
143489352 file3
143489354 output
```

Lo que se puede ver en el comando es la cantidad de bytes que ocupa cada archivo, y si se hace la diferencia entre ambos valores, obtenemos que difieren en 2 bytes. Esto podría deberse a una multitud de razones, pero la hipótesis que se plantea es que se debe a que se agregó “\r\n” al final. Veamos si esto es cierto; para ello ejecutamos `diff -u file3 output`:



```
--- file3      2023-06-19 16:28:37.164905579 -0300
+++ output    2023-06-19 16:48:12.325707488 -0300
@@ -1839606,3 +1839606,4 @@
a6TF7+340eJrY0+g7jSJXjp3q/ktYMP6Eb33VKdUEZEd+2SkeJhQlFhxPTcNo0T1lL/HNeek/0u
eGl6zy1PBhpZR+Wb8VhqXwd0v8+EX20LRYekgoP9GpUNB0gesg9orxqEQXidMItXYG43JDyRwE6U
VQ==
+
```

Imagen 6.4. Output de ejecutar `diff -u file3 output`.

Como se puede ver en la Imagen 6.4, la diferencia entre ambos comandos está en la última línea y se debe, de nuevo, al “\r\n”.

Consecuentemente, podemos concluir que las irregularidades que se observan al utilizar `curl` para obtener los mensajes del servidor se deben a la utilización de dicho comando.

## 7. Guía de instalación

1. Correr `make all` en la carpeta raíz del proyecto.
2. En la carpeta `bin` se habrán generado los archivos `server` y `client`.
3. Para correr el server se debe correr: `./server [-p <port>] -d <mail_dir> -a <user>:<pass> [[-u <user>:<pass>]...]` donde:
  - a. `-p` es el puerto donde se correrá el servidor. Si no se especifica, su valor por default es 1110



- b. *-d* es el directorio de mails del servidor. Aquí dentro se encontrarán los directorios de los distintos usuarios
- c. *-a* es el usuario y contraseña del administrador del servidor. Estas credenciales son las que se deben mandar siempre que se manden requests con el cliente desarrollado.
- d. *-u* son los distintos usuarios que hay en el servidor. En esta opción se deben incluir TODOS los usuarios que aparecen dentro del directorio mails y asignarles una contraseña.

## 8. Instrucciones para la configuración

---

Utilizar la aplicación de cliente (monitoreo) es simple. Se puede ejecutar de dos formas:

1. Si se desea imprimir una guía de uso, se debe correr: `./client -h`
2. Si se desea ejecutar un comando que será enviado al servidor, se debe correr: `./client -a <username>:<password> <command> [<args>]...` donde:
  - a. *-a* es el usuario y contraseña del administrador del servidor. Estas credenciales deben ser previamente especificadas al correr el server.
  - b. *command* es el comando que se quiere ejecutar siguiendo las especificaciones del RFC de Popcorn Protocol. Los comandos son los mismos que se especifican en el RFC ([ver sección 1.2.4](#)) Adicionalmente, el comando podría requerir argumentos (*args*).

Es importante otorgar los parámetros en el orden en el que se indican anteriormente.

Además, la aplicación cliente maneja internamente la versión del Popcorn Protocol que será enviada (en este caso 1) y el ID del pedido ([ver sección 1.2.2](#)).

Una vez que se reciba la respuesta del servidor, la aplicación cliente imprimirá en salida estándar la interpretación del código de estado y el contenido de la respuesta si es que se devuelve alguna.

Los códigos de estado se especifican en el RFC de Popcorn Protocol ([ver sección 1.2.3](#))

Finalmente, el programa finalizará exitosamente. Si se desea correr otro comando, se debe ejecutar el programa nuevamente de la manera descrita anteriormente.

## 9. Ejemplos de configuración y monitoreo

---

A continuación se mostrará, utilizando la aplicación cliente, algunos ejemplos de configuración y de monitoreo del servidor a través del protocolo desarrollado.

- Un usuario nueva intenta usar la aplicación cliente pero como no sabe qué comandos y opciones existen, se ejecuta solo el archivo `./client` (ver Imagen 9.1.a)

```
~/repos/popcorn master*  
> ./bin/client  
client: missing arguments  
Try "./client -h" for more information.
```

*Imagen 9.1.a. Ejecución de ./bin/client.*

Entonces, se espera que el usuario nuevo ejecute `./bin/client -h` para obtener más información acerca de los comandos. El resultado se muestra en la Imagen 9.1.b.

```
~/repos/popcorn master*  
> ./bin/client -h  
POPCORN protocol client  
Usage: ./client -a <user>:<password> <command> [args]  
  
Commands:  
  bytes  
  history  
  current  
  password <user> <password>  
  delete <user>  
  conc <max_users>
```

*Imagen 9.1.b. Ejecución de ./bin/client -h.*

- El cliente ejecuta el comando `bytes` exitosamente. En este caso se muestra el valor 0 porque nunca hubieron clientes conectados

```
~/repos/popcorn master*  
> ./bin/client -a admin:pass bytes  
response: OK  
value: 0
```

*Imagen 9.2. Ejecución del comando bytes.*

- El cliente ejecuta `history` exitosamente. En este caso el valor que muestra es 2 porque se conectaron en algún momento 2 clientes.

```
~/repos/popcorn master*  
> ./bin/client -a admin:pass history  
response: OK  
value: 2
```

*Imagen 9.3. Ejecución del comando bytes.*

- El cliente ejecuta *current* exitosamente. En este caso se muestra el valor 4 porque había 4 clientes conectados simultáneamente.

```
~/repos/popcorn master*  
> ./bin/client -a admin:pass current  
response: OK  
value: 4
```

*Imagen 9.4. Ejecución del comando current.*

- El cliente ejecuta *password* para cambiar la password del usuario “nico”. Como el usuario existe la respuesta es exitosa.

```
~/repos/popcorn master*  
> ./bin/client -a admin:pass password nico mypass  
response: OK
```

*Imagen 9.5. Ejecución del comando password.*

- El cliente ejecuta *conc* para poner como máximo 2 usuarios.

```
~/repos/popcorn master  
> ./bin/client -a admin:pass conc 2  
response: OK
```

*Imagen 9.6. Ejecución del comando conc.*

-

## 10. Documento de diseño del proyecto

---

En la presente sección se presentan varios diagramas para ayudar a comprender el flujo que sigue el servidor implementado. Se brindarán breves explicaciones acerca de lo observado en las imágenes.

En la Imagen 10.1 se muestra la inicialización del *server\_adt*. Luego de ser inicializado, el *server\_adt* se suscribe para leer en el selector. Nótese también que, en caso de recibir una conexión, se inicializa una *session*. Esta estructura guarda toda la información relevante para una sesión del cliente.

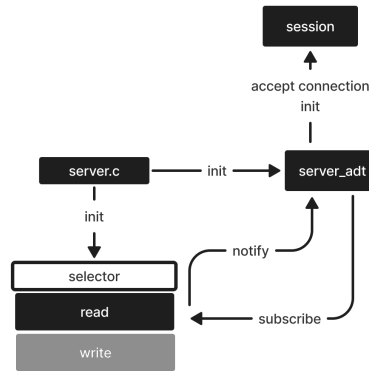


Imagen 10.1. Inicialización de `server_adt`.

La Imagen 10.2 muestra lo que sucede luego de que un cliente se conecta al servidor. Como se puede observar, la sesión pasa inmediatamente a escribir el greeting al usuario. Nótese las dos estructuras dentro de `session`: `action_stack` y `write_b`.

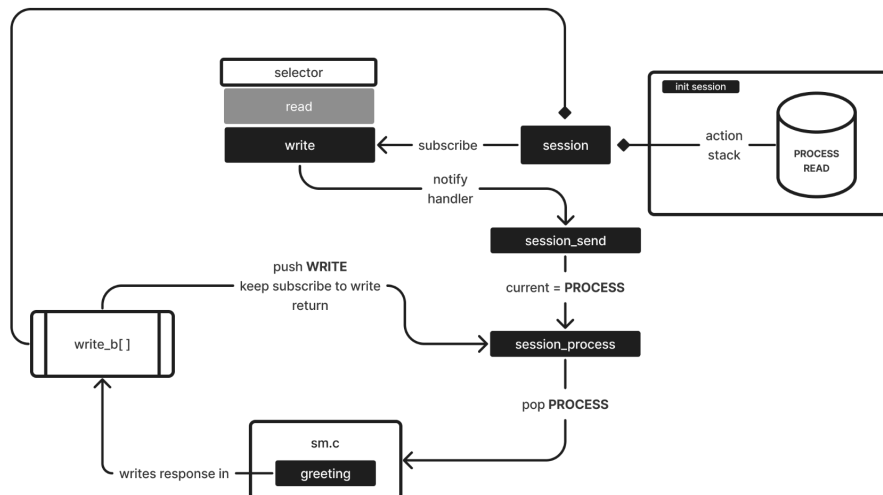


Imagen 10.2. Envío de `greeting`.

La Imagen 10.3 muestra qué pasa cuando se quiere escribir al usuario el `greeting`. Nótese que la pila comienza con `WRITE` y `READ`, por lo que en caso de que no se pueda enviar todo no se poppea del stack la acción de `WRITE`, cosa que sí sucede en el caso contrario.

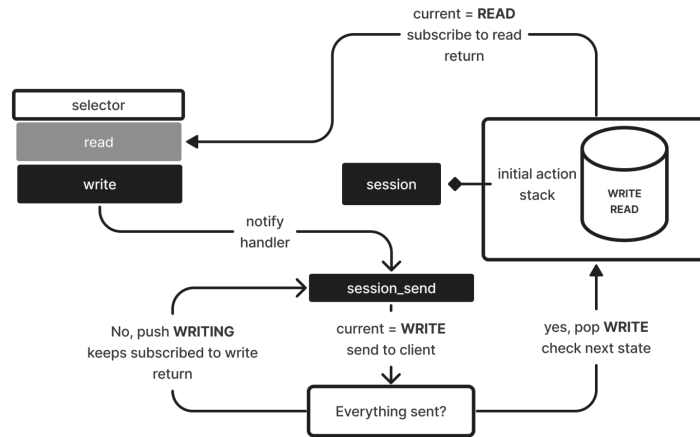


Imagen 10.3. Escritura al cliente.

En la Imagen 10.4 se puede ver todo lo que sucede luego de que el cliente pasa la etapa de greeting. Aquí se puede ver que el servidor hace lectura de lo que envía el cliente, y el manejo de los estados para poder devolver una respuesta adecuada.

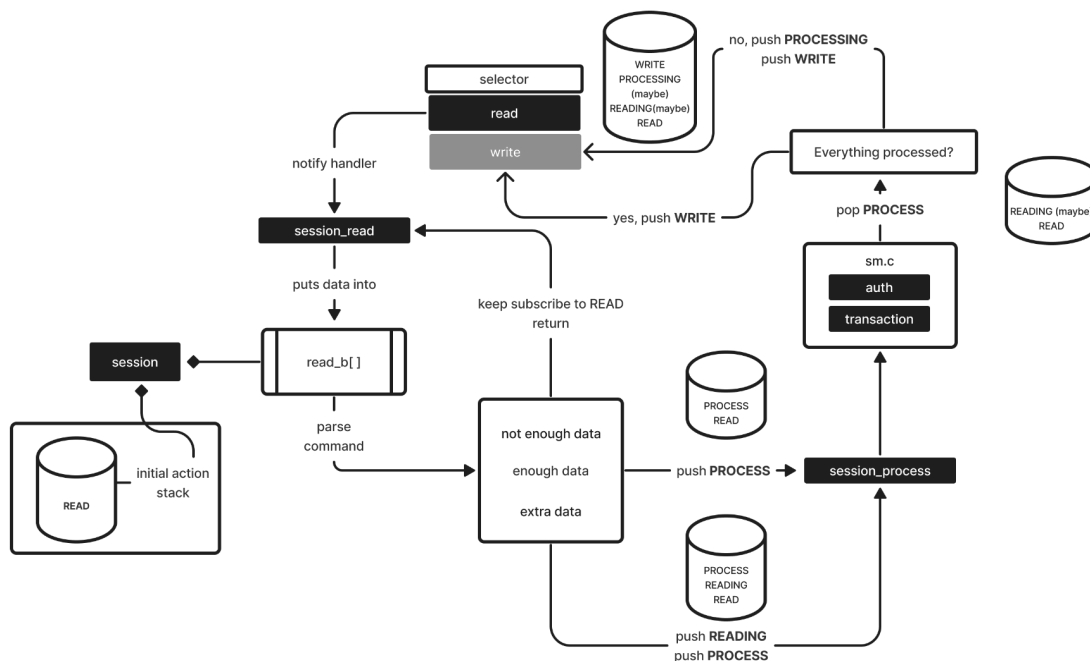


Imagen 10.4. Manejo de estados luego de greeting.

## 11. Referencias

[RFC 1939] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, May 1996.

[RFC 2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.