Informe TPE Protocolos de Comunicación

Alumnos:

- Occhipinti, Abril 61159
- Ranucci, Santino Augusto 62092
- Rossi, Victoria 61131
- Zakalik, Agustín 62068

Abstracto

En el siguiente documento se detalla el desarrollo del Trabajo Práctico enfocado en la implementación de un servidor para el protocolo POP3. El objetivo principal de este proyecto es implementar un servidor que cumpla con el protocolo POP3 (Post Office Protocol 3) y todas sus funcionalidades asociadas. Además, se incorporan características adicionales que permiten la administración y obtención de información relevante desde el servidor. Durante el desarrollo del trabajo práctico se exploran los aspectos fundamentales del protocolo POP3, se diseña e implementa un servidor eficiente y seguro, y se llevan a cabo pruebas exhaustivas para garantizar su correcto funcionamiento.

Índice

Descripción detallada de los protocolos y aplicaciones desarrollados

POP3

La implementación del servidor POP3 se realizó siguiendo las directrices del RFC 1939, lo que permitió establecer una comunicación adecuada entre el servidor y los clientes que se conectan a él. Se desarrollaron las funcionalidades esenciales del protocolo, tales como la autenticación del usuario, la descarga de mensajes y la manipulación de la bandeja de entrada.

Además, se incluyeron mejoras y extensiones basadas en el RFC 2449, que permiten la implementación de capacidades adicionales. Entre estas capacidades se encuentran:

1. **Capacidades (CAPA)**: El servidor implementa la funcionalidad CAPA, que permite a los clientes obtener información sobre las capacidades específicas compatibles con el servidor. Esto facilita la comunicación y la negociación de características adicionales entre el cliente y el servidor.
2. **Pipelining**: Se implementó la técnica de pipelining, que permite a los clientes enviar múltiples comandos al servidor sin esperar una respuesta antes de enviar el siguiente. Esto mejora significativamente la eficiencia y el rendimiento de la comunicación entre el cliente y el servidor, al reducir el número de intercambios de mensajes.

Con estas implementaciones, el servidor POP3 desarrollado proporciona una experiencia mejorada para los usuarios al brindar una comunicación más eficiente y un conjunto ampliado de capacidades. Dentro de las capacidades de nuestro servidor, se puede atender a un número de 500 clientes a la vez, los cuales pueden conectarse ya sea con IPV4 o IPV6.

MP3P (Monitor Pop3 Protocol)

El protocolo diseñado y desarrollado para ofrecer un servicio de monitoreo y administración al servidor POP3 es un protocolo a nivel aplicación que corre sobre UDP.

Los headers que se diseñaron para la comunicación cliente-servidor se detallan a continuación:

Request

```
MP3P <Version>\n
<requestId>\n
<authorization>\n
<command>
```

Response

```
MP3P <Version> <statusCode>\n
<requestId>\n
<data>
```

Donde,

`<Version>` es la versión de MP3P usada, en nuestro servidor, V1.0

`<requestId>` es un string alfanumérico aleatorio de 32 bytes generado por el cliente para identificar su datagrama, el servidor contesta con el mismo request Id.

`<authorization>` es una clave proporcionada por administrador de servidor, un string alfanumérico de 32 bytes.

`<statusCode>` puede ser un código de éxito o un código de error en caso de que el datagrama esté bien formado pero sea inválido.

Los códigos de errores son de la forma: `1xx` y los posibles códigos de errores son:

- 100: Unauthorized (El header <Authorization> del datagrama es inválido)
- 101: Version Mismatch (La versión no matchea)
- 102: Resource Not Found (El recurso al que se intento acceder es inexistente)
- 103: Bad authorization Formation (El campo de autorización esta mal formado, no es una cadena alfanumérica de 32 bytes)
- 104: Invalid Command Formation (El commando a ejecutar esta mal formado o es inexistente

En caso de éxito el statusCode es de la forma `2xx`

- 200 OK

El tamaño máximo de contenido que soporta el protocolo en la versión V1.0, es de 8096 bytes

Comandos del cliente de monitoreo

BT

Muestra los bytes transferidos

```
>BT
```

BR

Muestra los bytes recibidos

```
>BR
```

CC

Muestra la cantidad de conexiones concurrentes en el momento

```
>CC
```

HC

Muestra la cantidad de conexiones históricas

```
>HC
```

AU

Agrega un usuario

```
>AU <nombreUsuario> <claveUsuario>
```

MP

Modificar la contraseña

```
>MP <nombreUsuario> <claveNueva>
```

DU

Borrar un usuario

```
>DU <nombreUsuario>
```

LU

Listar los usuarios

```
>LU
```

LC

Listar las capabilidades

```
>LC
```

Idempotencia en MP3P

El protocolo MP3P es idempotente y las respuestas del servidor corresponden en base a si la acción que quizo hacer el usuario tomo efecto o no. A continuación se detalla lo que debe responder el servidor ante distintas situaciones considerando un datagrama bien formado:

AU

- 200 OK siempre, si el usuario no existía se debe crear, si el usuario ya existía actúa igual que MP, porque la acción que se deseaba tomo efecto.

LU

- 200 OK Siempre.

LC

- 200 OK siempre.

DU

- 200 OK siempre, si el usuario ya estaba entonces se elimino, si el usuario no estaba también 200 OK porque la acción tomo efecto en el servidor.

MP

- 200 OK Si el usuario existía pues se pudo modificar su contraseña
- 102 Resource Not Found si el nombre del usuario es inexistente

Errores:

Ante datagramas mal formados (es decir, superan el máximo de bytes por pedido permitido que es 8k) o bien, que no posean un request id de 32 bytes alfanumeric serán descartados por el servidor.

Los datagramas que respeten esto se pueden encontrar con los siguientes errores

- 100 Unauthorized: La clave de 32 bytes no matchea con el datagrama
- 101 Version Mismatch: La version del datagrama no es consistente con la del servidor (V1.0)
- 103 Bad authorization formation: La autorización no esta bien formada, no es una cadena alfanumérica de 32 bytes
- 104 Invalid Command Formation: El comando es inválido o bien, los argumentos pasados al mismo no corresponden

Decisiones de diseño tomadas

Logger

Para el sistema de logs, se opto por usar el código provisto por la cátedra, que se encuentra en la carpeta logger. El mismo provee la macro `log`, que permite establecer un nivel de log (como de

información, advertencia y error) y un mensaje personalizado, así como mostrar el número de línea en el cual se hizo el log.

Makefile y compilador

Para la generación del binario, se usaron las herramientas Makefile y CC. Makefile nos permitió generar un script de compilación versátil y mantenible en el tiempo, así como muy configurable.

Se utilizo el estándar de C C11, y se opto por agregar una serie de flags al compilador para sacarle el mayor provecho. Dentro de las flags agregadas, las más importantes son por un lado el conjunto de `-Wall, -pedantic, -Wextra`, que generan una gran cantidad de warnings y permiten producir un código de mayor calidad. Otro conjunto de flags importante fue `fsanitize=address -fsanitize=undefined -fsanitize=leak`, el cual se encarga de avisar ante cualquier perdida de memoria que se encuentre.

En cuanto al compilador, el uso de CC permite que el compilador utilizado sea fácilmente intercambiable (tan solo hace falta cambiar esa variable de entorno). Esto logra que el Makefile funcione en muchas mas computadoras, ya que no se requiere de tener determinado compilador instalado para generar el binario del proyecto.

Clientes

Dentro de la carpeta clients, se encuentran las funciones para el manejo de clientes POP3 de nuestro servidor que consisten en la apertura, escritura, lectura y cerrado de los mismos. Se decidió crear un struct de tipo `user_data`, que contiene toda la información relevante para el manejo de un cliente. Dado que el servidor permite 500 conexiones como máximo, se opto por crear un array de 500 posiciones, en la cual cada posición es de tipo `user_data`.

Aunque en un momento se discutió intercambiar este array por una estructura de datos más eficiente en memoria, como una lista dinámica, esta opción se descarto rápidamente. Esto se debe a que utilizar una lista dinámica en vez de un array implicaba un gran numero de syscalls adicional (`malloc` y `free`), lo cual tendría mas posibilidades de fallos y conlleva menos eficiencia temporal.

Dado que el servidor es no bloqueante, se manejan las 500 posibles conexiones simultáneas mediante el uso de la system call `select`, que permite saber qué file descriptors están listos para ser escritos o leídos de manera no bloqueante.

Obtención de los correos guardados

Los correos se guardan en subdirectorios dentro del directorio `mails`. Cada uno de estos subdirectorios pertenece a un usuario, por lo que si el usuario “jose” quiere obtener sus mails, se accederá a la carpeta `./mails/jose/`. Se utilizaron las funciones `fopen`, `fread` y `fclose` para el manejo de los correos, y la función `stat` nos permite conocer información útil de cada uno de estos, tales como el tamaño (que se utilizara en los comandos POP3 `stat` y `list`).

Una decisión importante tomada con respecto a los correos, fue la creación de un `struct mailsCache`. Esta estructura almacena información de todos los correos de cada usuario que tiene una sesión activa en el sistema, incluyendo el path a cada correo, su tamaño y el número que se le asignó. Se tomo esta

decision a causa de que la función `readdir`, que utilizamos para iterar por el directorio de correos de cada usuario, no asegura que un directorio se itere siempre en el mismo orden, por lo que surgieron problemas como que `list` no podia ser idempotente.

Parser de POP3

Se opto por crear una lista de comandos por cada usuario activo. Esta lista de comandos se va generando a medida que se lee el input del usuario, y es consumida a medida que `select` indica que se puede escribir al usuario, por lo que el servidor ejecuta el comando más viejo en la lista de comandos. Esta lista funciona como una cola, ya que los nuevos comandos se van agregando al final, y cuando se consume un elemento se devuelve el primero (que es siempre el más viejo que todavía no terminó de ejecutarse). Cada comando en esta lista posee un estado, el cual es señalado por un enum e indica si el comando se terminó de parsear y es válido, si se detecto como inválido, o si todavía falta recibir input para marcarlo como completo. El enum es el siguiente:

```
typedef enum command_status {
    WRITINGCOMMAND, //se esta escribiendo el comando (el siguiente input se concatenara al buffer de comando)
    WRITINGARG1,
    WRITINGARG2,
    COMPLETE, //el comando se terminó de parsear y es válido (esta listo para ser ejecutado)
    INVALID, //el comando es inválido pero todavía no llego un CRLF (hay que descartar lo que siga)
    COMPLETEINVALID, //el comando ya se terminó de parsear y es inválido (debe ser liberado)
} command_status;
```

MP3P

Para la implementación del protocolo MP3P del lado del servidor, se opto por utilizar un patrón **strategy.** Cuando un usuario envía un datagrama al servidor, este mismo se parsea y según el comando, se asigna un puntero a función al commando a correr. En cuanto al parser utilizado para esto mismo, se programo un autómata que parsea caracter a caracter el datagrama entrante. Si es inválido lo descarta, si el datagrama tiene algún problema que no sea de formación, el automata devuelve un estado que representa un puntero a función a ejecutar en base al error detectado, y si el datagrama no posee errores, el automata devuelve un estado que repersenta un puntero a función a ejecutar para completar el pedido del usuario.

Users

Para tener registro de los usuarios que tiene el servidor en memoria y de quienes están conectados y quienes no, se optó por utilizar el patrón **singleton.** Entonces en memoria se tiene una única instancia de una lista encadenada de usuarios de los cuales el servidor tiene registro, a su vez, esa única instancia en memoria guarda registro de que usuarios están conectados y cuales no. Al utilizar un patrón singleton, se tiene una única fuente de verdad en cuanto al acceso a los datos de los usuarios y también se centraliza este mismo.

Stats

Para controlar las estadísticas del servidor, se optó por utilizar nuevamente un patrón **singleton.** Para tener centralizado el registro y acceso a los datos a lo largo de los distintos módulos del programa. Este singleton consiste en una estructura que guarda los bytes transferidos, recibidos, conexiones concurrentes y conexiones históricas.

Problemas encontrados durante el diseño y la implementación

El primer problema encontrado fue a la hora de intentar escuchar para direcciones IPV4 e IPV6. Este problema fue resuelto luego de varios intentos, optando por utilizar una flag que permitía el uso de ambos estándares.

La creación del parser de POP3 también fue un problema, ya que era de una complejidad alta. Aunque finalmente no se implemento el comando TOP, nuestro parser tiene la capacidad de recibir hasta dos argumentos de 3 o 4 caracteres.

Finalmente, hubo varias complicaciones a la hora de programar RETR. Algunas de las cosas a tener en cuenta fueron que la system call `readdir` no asegura que un directorio se lea siempre en el mismo orden (por lo que hubo que encontrar alguna forma de asignarle un número a cada mail), y se tuvo que implementar una máquina de estados para que los mails devueltos tengan el byte-stuffing hecho correctamente.

Limitaciones de la aplicación

Una de las limitaciones más importantes que tiene el proyecto es que es single-threaded. Esto quita la posibilidad de sacarle todo el provecho al procesador de la computadora que corre el servidor POP3. Si se quisiera usar comercialmente, la primera mejora a hacerle sería convertirlo en multi-threaded (sin dejar de ser no-bloqueante para cada thread).

Otra de las limitaciones del proyecto es que el uso de la función `select` restringe el numero de file descriptors a escuchar, por lo que nunca se podría tener mas de 1000 clientes (aproximadamente) o más de 500 si se decide ejecutar algún proceso de transformación de correos. En el caso de este trabajo práctico, no se llegó a implementar dicho mecanismo de transformación, por lo que también se puede considerar una limitación del mismo.

Posibles extensiones

Para empezar, implementaríamos todas o un gran número de las funcionalidades que le añade el RFC 2449 al RFC 1939. Dentro de estas, se encuentran varios métodos de autenticación y comandos adicionales para poder hacer un uso más eficiente y completo del servidor a la hora de recuperar o modificar los correos de un usuario.

Finalmente, modificaríamos el servidor para que pueda correr de manera multithreaded. Esto sumaría en gran medida a la eficiencia del mismo.

Conclusiones

El desarrollo del trabajo práctico fue un gran desafío, en el cual no solo aprendimos sino que también pudimos poner en práctica todo lo visto en la materia a lo largo del cuatrimestre. El producto de esto fue un servidor para el protocolo de POP3 que incluye mecanismos de monitoreo que permiten recolectar métricas del sistema, y que permiten a los usuarios cambiar la configuración del servidor en tiempo de ejecución.

Ejemplos de prueba

Pruebas básicas

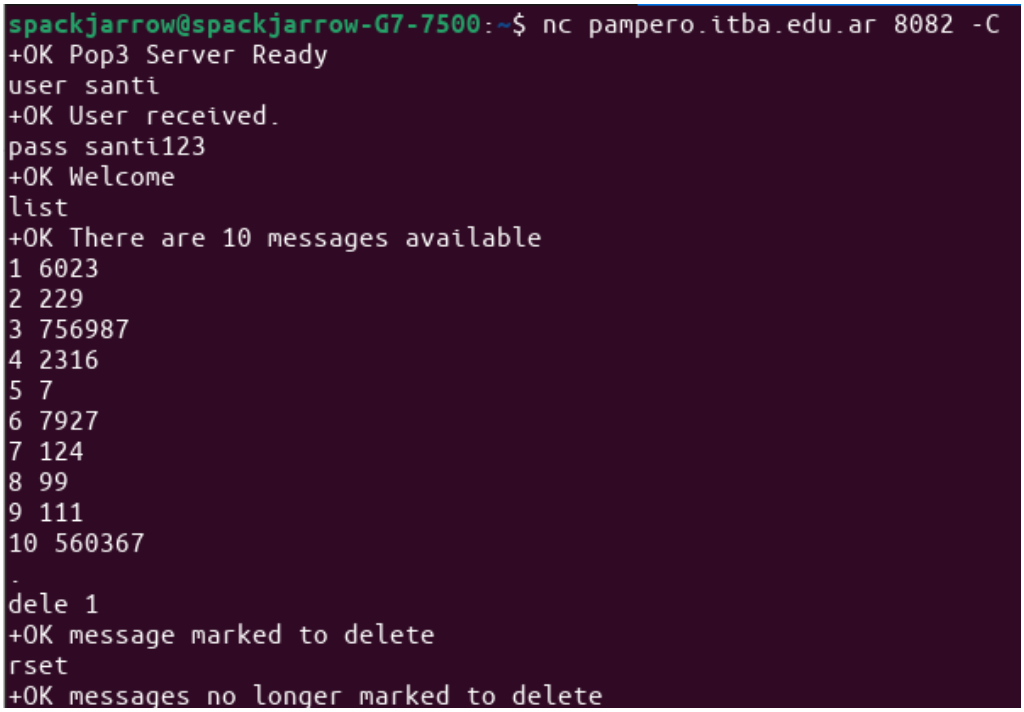
- Ejemplo de correr el servidor para que escuche en el puerto 8082 y agregue el usuario “santi” con la contraseña “santi123”

```
spackjarrow@spackjarrow-G7-7500:~/Documents/GitHub/Network-Protocols-POP3$ ./bin/proj
ect 8082 -u santi:santi123
INFO: src/server/serverUtils.c:172, Binding to :::8082
INFO: src/server/serverUtils.c:74, Binding to 0.0.0.0:6000
IPv4 UDP socket bound
INFO: src/server/serverUtils.c:74, Binding to :::6001
IPv6 UDP socket bound
```

- Ejemplo de conexión al servidor abierto anteriormente

```
spackjarrow@spackjarrow-G7-7500:~$ nc localhost 8082 -C
+OK Pop3 Server Ready
user santi
+OK User received.
pass santi123
+OK Welcome
list
+OK There are 11 messages available
1 1
2 6023
3 229
4 756987
5 2316
6 7
7 7927
8 124
9 99
10 111
11 560367
.
quit
+OK exiting
```

- Ejemplo de conexión al servidor corriendo en pampero



Una vez terminado el TPE, se dedicaron los últimos días a hacer tests sobre el mismo. Esto nos permitió estar seguros de que pudiera soportar 500 conexiones simultáneas en la vida real, y que no hubiera problemas con algunas funcionalidades como el pipelining o el byte-stuffing.

Pruebas de byte-stuffing

Gracias al script de python `client.py`, nos dimos cuenta de que existía un error en cuanto al byte-stuffing en un caso muy particular. Sin embargo, una vez detectado se corrigió el error rápidamente. Se crearon varios correos de prueba (con distintos formatos y el ascii “.” colocado en distintos lugares) y no se detectaron nuevos errores con dicho script.

Correr el servidor en pampero generando 500 conexiones simultáneas

Se utilizo el script de python `client.py` para simular un cliente que se conecta y hace infinitos requests. Para la creación del mismo se utilizo la librería `poplib`, que provee funciones predefinidas que generan pedidos POP3 válidos. La misma librería también se encarga de quitar el byte-stuffing a las lineas recibidas como respuestas a un `retr`, por lo que esto nos permitió probar que el byte-stuffing este funcionando correctamente.

Para poder generar X conexiones simultáneas, se puede utilizar el script `runMultipleClients.sh`. El mismo recibe un único argumento, que es la cantidad de procesos `client.py` que se desean crear.

Ambos archivos se encuentran en la carpeta `test` del proyecto.

Notar que este test se llevo a cabo habiendo desactivado el “Maildir lock”, que es parte del RFC 1939 e impide que se conecten multiples instancias de un mismo cliente. Si se corre este test con el “Maildir lock” activado, un solo cliente podrá autenticarse y el resto solo podrá conectarse (recibiendo un error al intentar la autenticación con el usuario ocupado).

Pruebas de velocidad para distintos tamaños de buffer

En esta prueba, se generó un archivo imprimible que ocupara un tamaño considerable. Para ello, primero se generó un archivo de 1GB, y luego se lo convirtió a un formato en el cual los caracteres sean imprimibles (base64). Para ello, se ejecutaron los siguientes comandos en la carpeta de mails del usuario “santi”.

```
dd if=/dev/urandom of=output.txt bs=1M count=1024 iflag=fullblock status=progress
base64 output.txt > giantBase64
rm output.txt
```

El archivo resultante se llama `giantBase64`, y su peso puede variar en cada ejecución de los comandos anteriores, pero es de aproximadamente 1.3GB.

Para medir los tiempos, se utilizo el comando `time nc localhost 8082 -C`. Se ejecuto el comando `retr` a este archivo con los tamaños de buffer 512, 1024, 2048 y 8096. Todos los resultados fueron de 3 minutos y 24 segundos, mas/menos 3 segundos.

Tener en cuenta que las respuestas tienen un error de uno o dos segundos (que es el tiempo tardado en escribir los comandos user, pass y retr).

Finalmente, decidimos dejar el tamaño del buffer en 512 bytes.

Guía de instalación

El primer paso a seguir es clonar el repositorio a un directorio en el que se tengan permisos de ejecución.

Server POP3

Para compilar el proyecto, se debe tener instalado un compilador de C, así como la herramienta Make. Los comandos a ejecutar son los siguientes:

```
make clean
make all
```

Tener en cuenta que para correr los comandos anteriores se debe estar posicionado en la carpeta padre del proyecto.

Esto genera un binario `project` en la carpeta bin y otro `client` dentro de la carpeta Client-Interface.

Instrucciones para la configuración

El trabajo no requiere de una configuración externa.

Ejemplos de configuración y monitoreo

Servidor POP3

El binario resultante se llama “project” y se encuentra en la carpeta “bin”. Sin embargo, para que funcione correctamente se debe correr desde la carpeta padre del proyecto como en el siguiente ejemplo

```
./bin/project {puerto_pasivo} -u {usuario1}:{contraseña1} -u {usuario2}:{contraseña2}
... -u {usuarioN}:{contraseñaN}
```

Reemplazar puerto_pasivo por el puerto en el que el servidor tiene que atender nuevas conexiones y reemplazar usuario y contraseña por los que se quiera agregar. Notar que el nombre de usuario debe coincidir con un nombre de directorio en la carpeta mails. De lo contrario, dicho usuario se podrá loggear al servidor pero no tendrá mails. Nótese que con -u se pueden agregar más usuarios.

Cliente de monitoreo

Ejecución:

En la carpeta Client-Interface correr el siguiente comando:

```
./client <IPv4|IPv6> <Server IP> <Server Port/Service> [Optional: -WV|-BA|-MA]
```

Donde `<Server Port/Service>` es 6000.

Opcionalmente, se puede agregar uno de los siguientes flags, para que se puedan ver las respuestas de errores del servidor :

-**WV (Wrong Version):** setea a todos los datagramas salientes con una version V2.0, que no es la del servidor, por lo tanto el servidor responderá siempre con un 101 Version Mismatch

-**BA (Bad Authorization):** setea todos los datagramas salientes con una autorización no correcta, el servidor siempre responderá un 100 (Unauthorized)

-**MA (Malformed Authorization):** setea todos los datagramas salientes con una autorización mal formada (es decir no una cadena de 32 bytes), el servidor siempre responderá con 103 (Bad authorization Formation)

Uso:

Aparece una interfaz de comandos tras ejecutar el programa

```
root@d504d46955d6:~/Client-interface# ./client IPv4 127.0.0.1 6000
Welcome! The commands available are:
1.  BT                       : To see the bytes that were transfered
2.  BR                       : To see the amount of bytes received
3.  CC                       : To see the current connections
4.  HC                       : To see the history connections
5.  AU <USER>                : To add a user
6.  CA <KEY>                 : To create a new key
7.  DM <METHOD>              : To disable a specific method
8.  MP <USER> <NEW PASSWORD> : To modify the password
9.  DU <USER>                : To delete a user
10. LU                       : To list users
11. HELP                     : To print options again
12. QUIT                     : To quit
>
```

BT (Bytes Transfered)

```
> BT
Sending bytes transferred request: ...

Response received: MP3P V1.0 200
rs3aEW9sbHWEMoUDpDQ3YjOejdhLivkH
48
```

Responde con un status 200 (OK), inmediatamente abajo el id del request, y luego los bytes enviados

BR (Bytes Received)

```
> BR
Sending bytes received request: ...

Response received: MP3P V1.0 200
V4RSfdaxxz024olEfKoIRRHY50V2zh2v
234
```

Responde con un status 200 (OK), inmediatamente abajo el id del request, y luego los bytes recibidos

HC (Historic Connections)

```
> HC
Sending historic connections request: ...

Response received: MP3P V1.0 200
nzNFBpjnZq0iLggllvcA4KPT2ZU89Uhx
0
```

Responde con un status 200 (OK), inmediatamente abajo del id del request, y luego las conexiones históricas (en este caso el pedido se hizo al iniciar el servidor)

CC (Concurrent Connections)

```
> CC
Sending concurrent connections request: ...

Response received: MP3P V1.0 200
ikoQK3NSrM1RiyslWP4mYoPzd3aa2RAj
0
```

Responde con un status 200 (OK), y inmediatamente abajo la cantidad de conexiones concurrentes del servidor.

LU (List Users)

```
> LU
Listing users...
Sending unknown command to server...

Response received: MP3P V1.0 200
poBAC4t2p7s6nReFCDsSh5HA1ODPlgSI
santi ranucci
```

Lista los usuarios y sus contraseñas

LC (List Capabilities)

```
> LC
Sending unknown command to server...

Response received: MP3P V1.0 200
z7aMo09Eo8PWqKtNhkGPpNMGGXzx64AD
MP3P V1.0
BT
BR
CC
HC
DU
AU
MP
LU
LC
```

Lista los comandos soportados por el servidor

AU (Add User)

```
> AU federico rodriguez
Sending Adding user request....

Response received: MP3P V1.0 200
Aal8HdOiFT5g7ON9JOZVV96A7fAtRskr


> LU
Listing users...
Sending Adding user request....

Response received: MP3P V1.0 200
axvznNvvR8IdQqLIbTPaqo82DaYpvbbD
federico rodriguez
santi ranucci
```

Agrega un usuario junto a su contraseña

DU (Delete User)

```
> DU santi
Sending Delete User request.....

Response received: MP3P V1.0 200
aGA1fSp8Ea6lsyX1IpAJ69PYCNWTSNl2


> LU
Listing users...
Sending Delete User request.....

Response received: MP3P V1.0 200
iZVhNEaGlOXwFHOCJiQKsKDIUiozwy0N
federico rodriguez
```

Elimina un usuario

MP (Modify Password)

```
> MP federico pepe
Sending modifying password request...

Response received: MP3P V1.0 200
ErXxIGxsItwnAw114iYiYbq2JzC1L7oq


> LU
Listing users...
Sending modifying password request...

Response received: MP3P V1.0 200
xBZeLhAz9Ml0PV3jkTUAT6qGVpemQW7l
federico pepe
```

Cambia la clave del usuario federico.

Si pasamos un usuario que es inexistente

```
Response received: MP3P V1.0 200
xr4y5w7oLRg8PgKPJhYWbevStbdAShwp
fede rodriguez
santi ranucci


> mp pepe pepito
Sending unknown command to server...

Response received: MP3P V1.0 104
pAINe3E6xexX6CwJOSliBymgGfOOihMH


> MP pepe pepito
Sending modifying password request...

Response received: MP3P V1.0 102
fMt5wxuosfmGJc1NUowRpQSirNQuoq8C
```

Vemos que si mandamos comando inválido (mp en minúscula) responde con un 104 (Invalid Command) y si intento cambiar clave a un usuario inexistente, responde con un (102) Resource not found.

Datagrama con -MA

```
INVALID IP VERSION, use IPv6 or IPv4
santino@santino-HP-Pavilion-Laptop-15-eh1xxx:~/Documents/GitHub/Network-Protocols-POP3/Client-interface$ ./client IPv4 127.0.0.1 6001 -M
A
Welcome! The commands available are:
1.  BT                       : To see the bytes that were transfered
2.  BR                       : To see the amount of bytes received
3.  CC                       : To see the current connections
4.  HC                       : To see the history connections
5.  AU <USER>                : To add a user
6.  MP <USER> <NEW PASSWORD> : To modify the password
7.  DU <USER>                : To delete a user
8. LU                        : To list users
9. LC                        : To list MP3P capabilities
10. HELP                     : To print options again
11. QUIT                     : To quit
> LU
Listing users....

Response received: MP3P V1.0 103
jlFi1PT310DaH7P64pkYb3ShIMS2XWGf
```

Responde 103 (Malformed Authorization)

Datagrama con -BA

```
santino@santino-HP-Pavilion-Laptop-15-eh1xxx:~/Documents/GitHub/Network-Protocols-POP3/Client-interface$ ./client IPv4 127.0.0.1 6001 -B
A
Welcome! The commands available are:
1.  BT                      : To see the bytes that were transfered
2.  BR                      : To see the amount of bytes received
3.  CC                      : To see the current connections
4.  HC                      : To see the history connections
5.  AU <USER>               : To add a user
6.  MP <USER> <NEW PASSWORD> : To modify the password
7.  DU <USER>               : To delete a user
8. LU                       : To list users
9. LC                       : To list MP3P capabilities
10. HELP                    : To print options again
11. QUIT                    : To quit
> LU
Listing users....

Response received: MP3P V1.0 100
ZiJE2EZUJxdM4e97MIXpX9tCUwlFLzTI
```

Responde con 100 (Unauthorized)

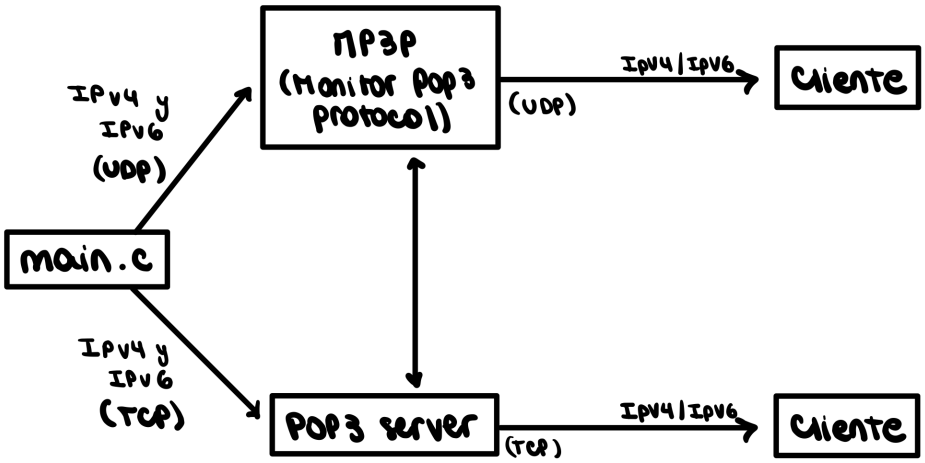
Datagrama con -WV

```
santino@santino-HP-Pavilion-Laptop-15-eh1xxx:~/Documents/GitHub/Network-Protocols-POP3/Client-interface$ ./client IPv4 127.0.0.1 6001 -W
V
Welcome! The commands available are:
1.  BT                      : To see the bytes that were transfered
2.  BR                      : To see the amount of bytes received
3.  CC                      : To see the current connections
4.  HC                      : To see the history connections
5.  AU <USER>               : To add a user
6.  MP <USER> <NEW PASSWORD> : To modify the password
7.  DU <USER>               : To delete a user
8. LU                       : To list users
9. LC                       : To list MP3P capabilities
10. HELP                    : To print options again
11. QUIT                    : To quit
> CC
Sending concurrent connections request: ...

Response received: MP3P V1.0 101
pri5ZmDJvmIauDx0lu7M7fjy6NkXBU4Z
```

Responde con 101 (Version Mismatch)

Documento de diseño del proyecto


MP3P
(Monitor pop3
protocol)
Ipv4 | Ipv6
(UDP)
Cliente
Ipv4 y
Ipv6
(UDP)
main.c
Ipv4 y
Ipv6
(TCP)
POP3 server
Ipv4 | Ipv6
(TCP)
Cliente