



---

**Trabajo Práctico Especial Protocolos de Comunicación:  
Informe del proyecto 1C 2023**

---

*Fecha de entrega: 23/06/2023*

*Alumnos:*

Juan Castro 62321  
Nicolás Rossi 53225  
Camila Sierra Pérez 60242  
Magdalena Flores Levalle 60077

*Docentes:*

Sebastian Kulesz  
Juan Francisco Codagnone  
Marcelo Fabio Garberoglio

*Grupo 5*

# Índice

<b>1. Introducción</b>	<b>1</b>
<b>2. Protocolos y aplicaciones desarrolladas</b>	<b>1</b>
2.1. Servidor POP3 . . . . .	1
2.2. Protocolo de management . . . . .	1
<b>3. Problemas encontrados durante el diseño y la implementación</b>	<b>3</b>
3.1. Entrada y salida constante del estado de read . . . . .	3
3.2. Parseando y ejecutando comandos en read . . . . .	3
3.3. Uso de enums dentro de las estructuras Request/Response del cliente . . . . .	3
<b>4. Limitaciones de la aplicación</b>	<b>3</b>
<b>5. Posibles extensiones</b>	<b>4</b>
5.1. Registro persistente de usuarios . . . . .	4
5.2. Server-Side rules . . . . .	4
5.3. Capa de sockets seguros (SSL) o Seguridad de la Capa de Transporte (TLS) . . . .	4
<b>6. Conclusiones</b>	<b>4</b>
<b>7. Ejemplos de prueba</b>	<b>5</b>
7.1. Comandos POP3 . . . . .	5
7.2. Filtro de salida . . . . .	7
7.3. Pipelining . . . . .	7
7.4. Sportar conexiones IPv4 e IPv6 . . . . .	8
7.5. Atender a múltiples clientes en forma concurrente y simultánea . . . . .	8
<b>8. Guía de instalación</b>	<b>10</b>
<b>9. Instrucciones para la configuración</b>	<b>10</b>
<b>10. Ejemplos de configuración y monitoreo</b>	<b>11</b>
<b>11. Documento de diseño del proyecto</b>	<b>14</b>

## 1. Introducción

El objetivo del Trabajo Práctico es la realización de un servidor para el protocolo POP3 (Post Office Protocol versión 3)[RFC1939] que pueda ser usado por MUA (Mail User Agents) tales como Mozilla Thunderbird, Microsoft Outlook y Evolution para la recepción de correos electrónicos. El mismo debe contar con requerimientos funcionales y no funcionales dictados por la cátedra y detallados a lo largo de este documento.

## 2. Protocolos y aplicaciones desarrolladas

### 2.1. Servidor POP3

El servidor pop3 es un servidor concurrente con entrada salida multiplexada no bloqueante. Para la implementación del mismo fue utilizado código aportado por la cátedra y el RFC 1939.

Su funcionamiento se basa en la creación de dos sockets pasivos, los cuales escuchan conexiones entrantes tanto para ipv4 como para ipv6. Además, se reserva otro socket (este no pasivo) para el protocolo UDP. Asimismo, el servidor tiene la opción de recibir opciones como parámetros por línea de comando, los cuales se implementaron respetando el estándar POSIX.

Para el manejo de las respuestas obtenidas, se optó por el uso de respuestas en el formato de POP3:

Estado	Descripción
+OK	Respuesta esperada
-ERR	Tipo de error

Además, pueden utilizarse los comandos de POP3:

- USER <username> para autenticar al usuario,
- PASS <password> para autenticar su contraseña,
- LIST <id> para devolver el tamaño del mail especificado,
- RETR <id> para devolver el mail con el id especificado,
- entre otros

### 2.2. Protocolo de management

Este protocolo permite a un administrador monitorear métricas, modificar algunos valores de configuración y detener el servidor. Para poder utilizarlo, debe pasarse como argumento al iniciar el servidor el token que se va a utilizar para la autenticación. Este debe ocupar 4 bytes.

Para realizar requests al servidor se optó por que el protocolo utilice un header de 10 bytes al principio del datagrama, donde se especifica la información del mensaje. Este es de tamaño variable y debe ser codificado en big-endian.

HEADER	DATA
10	Variable

DATA siendo los datos del mensaje. Este se utiliza para los argumentos necesarios por el comando usado. Puede estar vacío, estar formado por enteros sin signo de 8, 16 o 32 bits, o por strings null-terminated. Si el comando no necesita argumentos, este campo será ignorado.

El formato de este header es el siguiente:

VER	TYPE	CMD	ID	TOKEN
1	1	2	2	4

Donde:

- **VER** es es la versión del protocolo en uso.
- **TYPE** es el tipo de mensaje: 0x00 para una consulta, 0x01 para modificar.
- **CMD** es el comando a ejecutar.
- **ID** es el identificador del mensaje, debe coincidir con el de la respuesta.
- **TOKEN** es el token de autenticación.

El protocolo de manager utilizado en este proyecto es binario, no orientado a sesión el cual funciona sobre UDP. Toda la información acerca del protocolo puede encontrarse en el archivo `protocol.md` dentro de la carpeta docs.

A continuación se listan los comandos soportados por el cliente de management. Cabe aclarar que estos son case-sensitive, por lo que devolverá error si se escriben en mayúscula en vez del resultado esperado.

Comandos	Descripción
help	Imprime la info de los comandos.
exit	Salte del cliente.
list <page>	Retorna la página solicitada del directorio del usuario.
get-page-size	Retorna el número de usuario por página (máx 200).
connections	Retorna el número de conexiones concurrentes en el servidor.
bytes	Retorna el número bytes enviados por el servidor.
historic	Retorna el número de requests atendidos por el servidor.
add-user <username:password>	Añade un usuario nuevo al servidor.
del-user <username>	Borra un usuario del servidor.
set-page-size <page-size>	Setea un nuevo número de usuarios por página (máx 200).
stop	Detiene el servidor.

Toda la información sobre el protocolo se encuentra en el archivo `protocol.md` en la carpeta docs del trabajo.

### **3. Problemas encontrados durante el diseño y la implementación**

#### **3.1. Entrada y salida constante del estado de read**

Como estábamos saliendo y entrando constantemente del estado donde se lee el input del usuario, y hacíamos uso de una función que se llamaba al pasar de estado en la state machine que se encargaba de inicializar el parser cada vez que entrábamos a ese estado, teníamos el problema de que cuando el input no era suficiente para que el parser llegara a una conclusión se perdía lo que habíamos leído hasta el momento, por lo que no reconocíamos los comandos del cliente.

#### **3.2. Parseando y ejecutando comandos en read**

Al principio, a penas leíamos del input del usuario, parseábamos hasta dejar vacío el input buffer e íbamos ejecutando los comandos en el momento, pero esto generaba problemas no solo con PIPELINING, dado que después al pasar al estado de enviar, una vez que terminábamos nos poníamos con interés en read en el selector, pero el selector no nos llamaba hasta que había algo para leer, entonces si quedaba algún comando a medio leer no lo seguíamos parseando hasta que nos volvía a mandar algo el cliente, mientras que por otro lado estábamos procesando comandos sin saber si el usuario los iba a recibir, por lo que estaríamos gastando tiempo de procesador en nada.

Para resolver esto, se parsea hasta conseguir un comando en el read, y una vez que se ejecuta, se pasa al estado de write donde se envía la respuesta y después continuamos parseando el resto del input, y una vez que el comando termina volvemos a pasar por el selector para enviarlo, asegurándonos que los buffers no se llenen, y una vez que nos quedamos sin comandos para procesar o el parser no puede llegar a una conclusión porque falta el resto del comando, pasamos al estado de read para recibir más cosas del cliente.

#### **3.3. Uso de enums dentro de las estructuras Request/Response del cliente**

Al momento de probar el cliente de management el comportamiento era inconsistente, y al analizar los paquetes enviados y recibidos el tamaño no era el esperado. El problema estaba en que las estructuras hacían uso de enums en algunos campos, por ejemplo para la versión del protocolo que según lo establecido debe ser 1 byte, y ello ocasionaba que se asignase un espacio de memoria por defecto de un INT. La solución fue no utilizar enums en la estructuras request/response y sino que usar directamente los tipos 'uint8\_t', 'uint16\_t'.

### **4. Limitaciones de la aplicación**

Como estamos usando un pselect, tenemos un máximo de 1024 file descriptors que podemos monitorear, definidos en la macro FD\_SETSIZE. Teniendo en cuenta que uno de estos slots tiene que ser usado para el socket pasivo donde recibimos conexiones y otro para el que es usado para el socket de management, y luego cada usuario puede llegar a usar un file descriptor y puede llegar a usar otro para la lectura del archivo en caso de realizar un RETR, podemos atender hasta

511 clientes si todos leen un archivo, o 1022 si ninguno usa RETR.

## 5. Posibles extensiones

### 5.1. Registro persistente de usuarios

Con el diseño actual, no se cuenta con un registro de usuarios persistente. Las credenciales de los usuarios están guardadas en memoria, lo que ocasiona que ante una caída del servidor, se perderían todas las credenciales de usuarios. Como posible mejora, se puede emplear algún método de persistencia de credenciales, pudiendo ser desde un sencillo archivo que almacene dichas credenciales y las restablezca. O alguna otra estrategia mas sofisticada como una base de datos.

### 5.2. Server-Side rules

Como otra posible mejora se podría implementar la capacidad para que los usuarios establezcan reglas en el lado del servidor que puedan realizar acciones automáticamente en los correos electrónicos entrantes, como reenviar, eliminar o categorizar según ciertos criterios.

### 5.3. Capa de sockets seguros (SSL) o Seguridad de la Capa de Transporte (TLS)

Otra mejora importante es implementar SSL o TLS ya que agrega una capa de seguridad al cifrar la comunicación entre el cliente y el servidor. Esto evita el acceso no autorizado y protege la información confidencial, como los nombres de usuario y las contraseñas, para que no sean interceptadas. *Estos temas fueron presentados y comentados, pero escapan al scope de la materia y de este trabajo practico.*

## 6. Conclusiones

Para el desarrollo de este trabajo debió hacerse uso de conocimientos adquiridos en materias previas como Arquitectura de Computadoras y Sistemas Operativos. Además, se pudo poner en práctica los conceptos teóricos visto a lo largo de la materia este cuatrimestre. Los conceptos que más se destacaron fueron el uso de protocolos pop3 y el manejo avanzado de sockets.

## 7. Ejemplos de prueba

### 7.1. Comandos POP3

```
magui@maguiscpc:~/Desktop/TPE-Protos$ nc -C -v localhost 1110
Connection to localhost 1110 port [tcp/*] succeeded!
+OK POP3 server ready
USER
-ERR Invalid arguments
USER magflores
+OK
PASS password
+OK Logged in
LIST
+OK Listing mails
1 687
2 1717
.
```

Figura 1: Comandos USER, PASS y LIST

```
CAPA
+OK
USER
PIPELINING
IMPLEMENTATION TPE-Protos-G05-v01
.
```

Figura 2: Comando CAPA

```
RETR 1
+OK 687 octets
From: sarah.smith@example.com
To: mark.jones@example.com
Subject: Meeting Reminder

Hi Mark,

Just a friendly reminder about our meeting tomorrow at 2:00

Looking forward to seeing you there.

Best regards,
Sarah
```

Figura 3: Comando RETR

```
DELE 1
+OK Message 1 deleted
RETR 1
-ERR no such message
RSET
+OK Maildrop has 2 messages (2185 octets)
DELE 1
+OK Message 1 deleted
```

Figura 4: Comando RSET

```
LIST
+OK Listing mails
1 468
2 1717
.
DELE 1
+OK Message 1 deleted
RETR 1
-ERR no such message
```

Figura 5: Comandos DELE y RETR

```
LIST 2
+OK 2 1717
DELE 2
+OK Message 2 deleted
LIST 2
-ERR Mail already deleted
```

Figura 6: Comandos DELE y LIST

```
NOOP
+OK
```

Figura 7: Comando NOOP



## 7.2. Filtro de salida

Para realizar un prueba sencilla del filtro de salida haremos uso de un script bash que lee la entrada, descarta la primer linea leída y luego envía el resto a la salida.

Script: `remove_first_line.sh`

```
#!/bin/bash
#Read input and discard the first line
IFS= read -r first_line
while IFS= read -r line; do
    echo "$line$"
done
```

Debemos asegurarnos que el script tenga los permisos necesarios de ejecucion y que pueda ser llamado desde terminal.

```
sudo chmod +x remove_first_line.sh
sudo mv remove_first_line.sh /usr/local/bin/remove_first_line.sh
```

Para utilizarlo en el servidor hacemos uso del flag -f

```
./pop3server -d path/to/mails -u username:password -f remove_first_line.sh
```

Así, luego de hacer un **RETR** la primer linea del email sera descartada. Por ejemplo, para:

```
Message-ID: <9687359.1075840397534.JavaMail.evans@thyme>
Date: Tue, 5 Feb 2002 16:40:23 -0800 (PST)
From: infrastructure.ubsw@enron.com
To: canada.dl-ubsw@enron.com
Subject: Quick Tips for the UBSWE migration
Content-Type: text/plain;
```

Hellooo World!!

La salida sera:

```
Date: Tue, 5 Feb 2002 16:40:23 -0800 (PST)
From: infrastructure.ubsw@enron.com
To: canada.dl-ubsw@enron.com
Subject: Quick Tips for the UBSWE migration
Content-Type: text/plain;
```

Hellooo World!!

Notar que la primer linea sera descartada.

## 7.3. Pipelining

Para testear que nuestro servidor soporte pipelining, creamos un script en bash *pipelining.sh* que se encuentra dentro de la carpeta tests. El mismo ejecuta el servidor y crea al usuario

con su contraseña. Luego realiza un printf de los comandos en la conexión al servidor (realizada con netcat). El servidor luego devuelve la respuesta a cada pedido, no se queda solo con el último comando.

```
#!/bin/bash

cd /home/magui/Desktop/TPE-Protos/cmake-build-debug/src
./pop3server -d /home/magui/Desktop/TPE-Protos/.mails -u magflores:password

printf "USER magflores\nPASS password\nCAPA\nLIST\n" | nc -C -v 127.0.0.1 1110
```

Figura 8: Script

```
Connection to localhost 1110 port [tcp/*] succeeded!
+OK POP3 server ready
+OK
+OK Logged in
+OK
USER
PIPELINING
IMPLEMENTATION TPE-Protos-G05-v01
.
+OK Listing mails
1 687
2 1717
```

Figura 9: Terminal

#### 7.4. Sportar conexiones IPv4 e IPv6

La conexión IPv4 está demostrada con el testeo previo de pipelining, y para testear que soporte IPv6 simplemente realizamos el mismo testeo cambiando la dirección '127.0.0.1' por ':::1'. El mismo se encuentra en el archivo *ipv6.sh*.

```
#!/bin/bash

cd /home/magui/Desktop/TPE-Protos/cmake-build-debug/src
./pop3server -d /home/magui/Desktop/TPE-Protos/.mails -u magflores:password

printf "USER magflores\nPASS password\nCAPA\nLIST\n" | nc -C -v :::1 1110
```

Figura 10: Terminal

#### 7.5. Atender a múltiples clientes en forma concurrente y simultánea

Para esta sencilla prueba haremos uso de un script **add\_user.sh** que se encarga de la creación de usuarios 'dummy' para luego poder simular una cantidad de usuarios concurrentes.

Los usuarios creados son del tipo:  
 nombre de usuario: user*i*, *donde i* es el numero de iteracion en el for-loop.  
 contraseña: password , igual para todos los usuarios.

Una vez que el servidor ya se encuentre corriendo, al ejecutar el script add\_user.sh creara los usuarios utilizando el managerclient y además creara dentro del directorio de mails una carpeta para cada usuario.

**Notar que el directorio actual es /workspace/protos/.mails ya que ese fue el utilizado para ejecutar esta prueba con el servidor**

Una vez agregados los usuarios y creados los directorios, el script pasa a simular un inicio de sesión por cada uno de los usuarios. Para ello, se ejecuta varias veces el commando:

```
printf "USER user%s\nPASS password\n" $i | nc -C -v localhost 1110 &
```

y utilizando el '&' al final para que los procesos se ejecuten por separado.

Si todo salio correctamente, lo usuarios podrán logearse. y se mostrara el mesage de inicio de sesión exitoso por cada usuario.

```
#!/bin/bash
# Cant of users to be created
users=500
# Export token to be able to use the manager client
export TOKEN=12345678
# For-loop to add the users credentials to the server and exit
for ((i = 1; i <= $users; i++)); do
    ~/workspace/protos/TPE-Protos/cmake-build-debug/src/managerclient 127.0.0.1 9090 <<EOF
add-user user$i:password
exit
EOF
done

# For-loop to connect to the server and login with the users credentials
for ((i=1; i<=$users; i++))
do
    # Creat user's directory otherwise the login will fail
    mkdir ~/workspace/protos/.mails/user$i;
    # Run in a background process to simulate multiple users
    printf "USER user%s\nPASS password\n" $i | nc -C -v localhost 1110 &
done
```

Para eliminar las carpetas de los usuarios que fueron creados durante la prueba, basta con correr el siguiente script.

**Asegurarse que la cantidad de usuarios, y el directorio de mails sea el adecuado.**

```
#!/bin/bash

# Cant of users that were created with add_user.sh
```

```
users=500
# Remove folders created for this test
for ((i=1; i<=$users; i++));
do
    rm -rf ~/workspace/protos/.mails/user$i;
done

printf "done\n"
```

## 8. Guía de instalación

Una vez descargada la carpeta del trabajo (<https://github.com/JR-Castro/TPE-Protos>), debe posicionarse en el directorio principal e ingresar el siguiente comando:

```
cd cmake-build-debug/
```

Luego, debe dirigirse al directorio src y ejecutar el siguiente comando para compilar el servidor pop3:

```
./pop3server -d [path]/.mails -u [user]:[password]
```

Donde:

- **path** es el path donde se encuentran los mails a enviar.
- **user** es el usuario que se desea autenticar.
- **password** es la contraseña del usuario que se desea autenticar.

Por último, para conectarse al servidor pop3, debe ejecutar el siguiente comando:

```
nc -C -v localhost 1110
```

## 9. Instrucciones para la configuración

Se deben correr los scripts de CMake para cargar el proyecto y, por defecto, los MUA toman el puerto 1110, nuestro protocolo toma el puerto 9090 y la conexión pop3 al origin server se encuentra en el puerto 1010.

Los comandos soportados para hacer cambios a esta información se listan en la siguiente tabla:

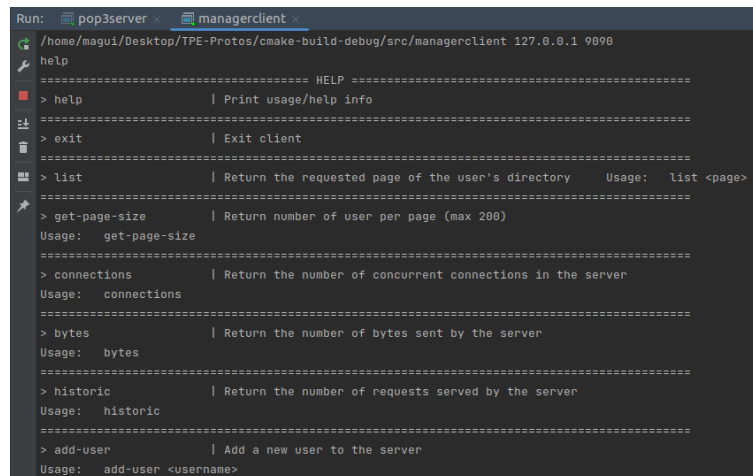
Comandos	Descripción
-h	Imprime la ayuda y termina.
-l <pop3 addr>	Dirección donde servirá el proxy POP3.
-L <conf addr>	Dirección donde servirá el servicio de management.
-p <pop3 port>	Puerto entrante conexiones POP3.
-o <conf port>	Puerto entrante conexiones management.
-u <user:pass>	Registra usuario y contraseña de un usuario válido.
-d <directory>	Directorio donde se almacenarán los mails.
-v	Imprime información sobre la versión y termina.

## 10. Ejemplos de configuración y monitoreo

A continuación se muestra el uso de los comandos soportados por el protocolo de management:

- **Comando help**

Imprime la información de los comandos soportados por el cliente de management.



```

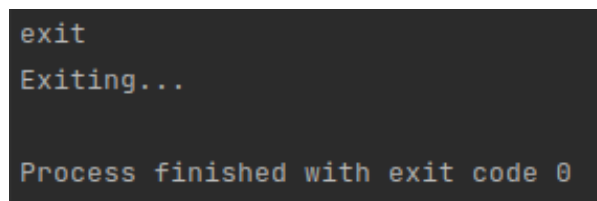
Run: pop3server x managerclient x
/home/magui/Desktop/TPE-Protos/cmake-build-debug/src/managerclient 127.0.0.1 9090
help
===== HELP =====
> help          | Print usage/help info
=====
> exit          | Exit client
=====
> list          | Return the requested page of the user's directory   Usage: list <page>
=====
> get-page-size | Return number of user per page (max 200)
Usage: get-page-size
=====
> connections   | Return the number of concurrent connections in the server
Usage: connections
=====
> bytes         | Return the number of bytes sent by the server
Usage: bytes
=====
> historic      | Return the number of requests served by the server
Usage: historic
=====
> add-user      | Add a new user to the server
Usage: add-user <username>

```

Figura 11: Utilización del comando help

- **Comando exit**

Cierra el cliente y sale.



```

exit
Exiting...

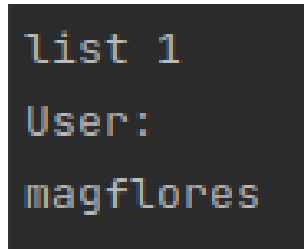
Process finished with exit code 0

```

Figura 12: Utilización del comando exit

- **Comando list**

Imprime la página solicitada del directorio del usuario. Debe pasarse como parámetro la página que desea retornar.

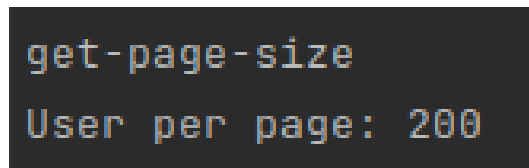


```
list 1
User:
magflores
```

Figura 13: Utilización del comando list

- **Comando get-page-size**

Retorna el número de usuario por página con un máximo de 200.

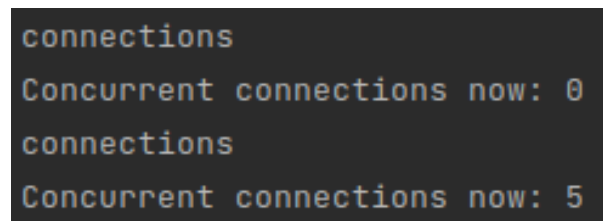


```
get-page-size
User per page: 200
```

Figura 14: Utilización del comando get-page-size

- **Comando connections**

Retorna el número de conexiones concurrentes en el servidor. Para la siguiente imagen se ingresó el comando primero sin conexiones al servidor y luego, utilizando el script del archivo `connections.sh` de la carpeta de tests, se generaron 5 conexiones al servidor y se volvió a correr el comando para obtener una salida distinta.

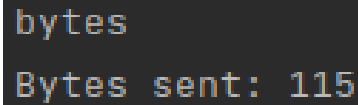


```
connections
Concurrent connections now: 0
connections
Concurrent connections now: 5
```

Figura 15: Utilización del comando connections

- **Comando bytes**

Retorna el número de bytes enviados por el servidor.

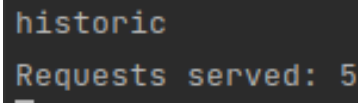


```
bytes
Bytes sent: 115
```

Figura 16: Utilización del comando bytes

- **Comando historic**

Retorna el número de requests atendidos por el servidor.

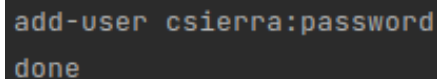


```
historic
Requests served: 5
```

Figura 17: Utilización del comando historic

- **Comando add-user**

Añade un usuario nuevo al servidor. Debe pasarse como parámetro el nombre del nuevo usuario con su contraseña, separado por ":".

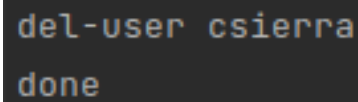


```
add-user csierra:password
done
```

Figura 18: Utilización del comando add-user

- **Comando del-user**

Añade un usuario nuevo al servidor. Debe pasarse como parámetro el nombre del usuario que se desea eliminar.

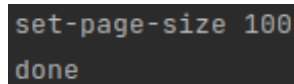


```
del-user csierra
done
```

Figura 19: Utilización del comando del-user

- **Comando set-page-size**

Setea un nuevo número de usuarios por página con un máximo de 200. Debe recibir por parámetro dicho número.



```
set-page-size 100
done
```

Figura 20: Utilización del comando set-page-size

- **Comando stop**  
Detiene el servidor.

```
stop
done
```

Figura 21: Utilización del comando stop

- **Manejo de errores**

En caso de ingresarse mal un comando, se retorna el mensaje de [ERROR] seguido del tipo de error que corresponda. Por ejemplo, en caso de no ingresar la contraseña al crear un usuario, se obtiene:

```
add-user user
[Error] Invalid argument.
```

Figura 22: Manejo de error en caso de que falte un parámetro

## 11. Documento de diseño del proyecto

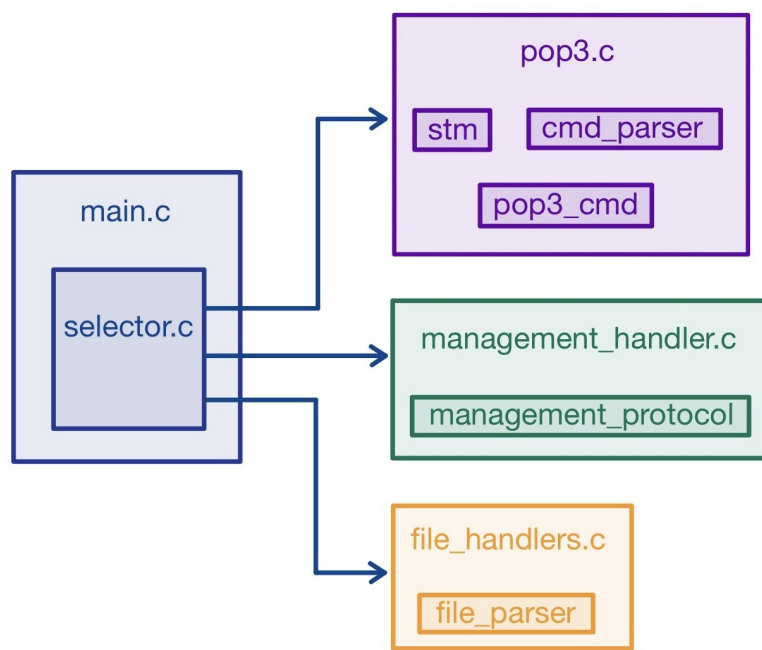


Figura 23: Diseño del flujo del proyecto