TP Protocolos de Comunicación

Alumno	Legajo	Mail
Arnaude, Juan Segundo	62184	jarnaude@itba.edu.ar
Canevaro, Bautista Ignacio	62179	bcanevaro@itba.edu.ar
Daneri, Matías Ezequiel	60798	mdaneri@itba.edu.ar
Wodtke, Matías	62098	mwodtke@itba.edu.ar

Tabla de contenidos

Descripción detallada de los protocolos y aplicaciones desarrolladas

Implementación del Protocolo POP3

El protocolo Post Office Protocol version 3 (POP3) es un protocolo de aplicación estándar para la recepción de correos electrónicos. En este proyecto implementamos un servidor POP3 siguiendo las especificaciones del RFC-1939. El servidor desarrollado incorpora las funcionalidades esenciales del protocolo POP3, lo que incluye una serie de comandos básicos y estados necesarios para una operación efectiva.

Comandos Básicos Implementados

- **USER**: Autenticación del nombre de usuario.
- **PASS**: Autenticación de la contraseña.
- **QUIT**: Finalización de la sesión.
- **STAT**: Obtención de estadísticas del buzón.
- **LIST**: Listado de mensajes en el buzón.
- **RETR**: Recuperación de mensajes específicos.
- **DELE**: Eliminación de mensajes.
- **NOOP**: Comprobación del estado del servidor.
- **RSET**: Resetear el estado de la sesión.

Estados Implementados

- **TRANSACTION**: Estado durante el cual el cliente puede solicitar acciones relacionadas con los mensajes.
- **AUTHORIZATION**: Estado inicial para la autenticación del usuario.

Características Adicionales

- Implementación de características avanzadas descritas en el RFC–2449, como el **pipelining** y el comando **CAPA**. Estas características mejoran la eficiencia y la funcionalidad del servidor.

Capacidad del Servidor

- El servidor tiene la capacidad de manejar hasta 512 clientes de manera concurrente, asegurando así un servicio robusto y eficiente.

Aplicación Cliente

- Se ha desarrollado una aplicación cliente que facilita la interacción con el servidor. Esta aplicación permite:
  - Acceder a estadísticas del servidor.
  - Modificar configuraciones del servidor, como el directorio de correos electrónicos.
  - Administrar cuentas de usuario, incluyendo la creación, modificación de contraseñas y ajustes de límites de correo.

Protocolo de comunicación entre cliente y servidor.

En el desarrollo de nuestro sistema cliente/servidor, se implementó un protocolo de comunicación específico diseñado para garantizar una interacción eficiente y clara entre la aplicación cliente y el servidor. Se define un formato de comando que es esencial para la correcta transmisión y recepción de instrucciones entre ambos componentes del sistema.

Cada comando que se ejecuta desde la aplicación cliente sigue un formato estructurado, diseñado para ser fácilmente interpretado por el servidor.

El formato general es el siguiente:

```
token [token_value] [command-name]|[left-value]:[right-value]
```

En caso de que el comando sea válido y aceptado por el servidor, este debe responder con el formato de mensaje siguiente.

```
+OK [msg]
```

Caso contrario, por formato inválido o comando no aceptado, debe responder con un mensaje con el siguiente formato.

```
-ERR [msg]
```

Operación básica

El funcionamiento del sistema comienza con la activación del servicio de escucha en el servidor, configurado por defecto en el puerto UDP 9090. Este puerto puede modificarse mediante los argumentos. Cuando un cliente quiere interactuar con el sistema, debe enviar comandos en el formato preestablecido.

Los comandos disponibles se categorizan de la siguiente manera, dependiendo de los argumentos necesarios:

1. **Comandos con un único argumento (*left-value*):** Estos comandos requieren solo un parámetro para su ejecución.
2. **Comandos con dos argumentos (*right-value*):** Estos requieren dos parámetros para funcionar correctamente.

En casos donde un argumento no es necesario para un comando específico, se debe utilizar la cadena **“not-given”** para mantener la integridad del formato.

La estructura de cada comando se detalla a continuación, resaltando su formato y los argumentos requeridos. Cada comando está diseñado para realizar una función específica dentro del sistema, permitiendo una interacción eficiente y adaptada a las necesidades del usuario.

```
token [token_value] help|not-given:not-given
token [token_value] add-user|[username]:[password]
token [token_value] change-pass|[username]:[new_password]
token [token_value] change-maildir|[new_maildir]:not-given
token [token_value] remove-user|[username]:not-given
token [token_value] not-given|not-given:not-given
token [token_value] version|not-given:not-given
token [token_value] get-max-mails|not-given:not-given
token [token_value] set-max-mails|[new_max_mails]:not-given
token [token_value] stat-historic-connections|not-given:not-given
token [token_value] stat-bytes-transferred|not-given:not-given
```

Implementación de Byte Stuffing en el servidor

Introducción

En el desarrollo de nuestro servidor POP3, una de las características claves implementadas fue el "byte stuffing". Esta técnica es esencial para garantizar una comunicación clara y sin ambigüedades entre el servidor y el usuario.

Necesidad

El byte stuffing se vuelve crucial debido a la forma en que el protocolo define el fin de un mensaje: la secuencia `"\r\n.\r\n"`. Si esta secuencia aparece dentro del contenido de un mensaje de correo electrónico, podría ser interpretada erróneamente como el fin del mensaje. Para evitar esta confusión y garantizar que los mensajes se transmitan correctamente, utilizamos el byte stuffing.

Implementación

La implementación se realizó siguiendo estos pasos:

1. **Detección de Secuencias**: Durante la preparación de los mensajes de correo electrónico, en la etapa de `RETR`, escaneamos el contenido del mail, a la vez que producimos su salida, en busca de cualquier instancia de la secuencia `"\r\n.\r\n"`.
2. **Aplicación**: Si se encuentra esta secuencia dentro del contenido del mensaje, insertamos automáticamente un byte adicional (en este caso, un punto '.') antes del punto en la secuencia. De este modo, `"\r\n.\r\n"` se transforma en `"\r\n..\r\n"`.

Problemas encontrados durante el diseño y la implementación

Un problema que encontramos en el desarrollo en el uso de comandos como el RETR fue que cuando el mail a leer era muy extenso, este llenaría el buffer (esto podría suceder en cualquier otro comando que escriba en consola, pero en este caso fue que lo notamos y en el que sucedería siempre). Cuando se intentaba escribir en el buffer sin haber espacio, la aplicación fallaba.

Para solucionar esto se optó por implementar un flag is_finished que simplemente se seteaba como falsa cuando al intentar devolver algo que no entrase en el buffer, la función sería re-ejecutada hasta que se imprima toda la información necesaria.

En otra instancia se encontró el problema de los comandos como CAPA o QUIT que debían ser aceptados tanto en estado de AUTHORIZATION como en el estado de TRANSACTION. En nuestra implementación se complicaba el hecho de que sea aceptable un comando para más de un estado, ya que al leerlo chequeamos sí el estado único posible para ese comando es igual al del estado actual. Para solucionar esto, simplemente se trató a los comandos que son multiestado como comandos distintos que apunten a la misma función de la siguiente manera:

```
struct command commands[] = {

    {.state = AUTHORIZATION,
     .name = "USER",
     .arguments = REQUIRED,
     .handler = user_handler},
    {.state = AUTHORIZATION,
     .name = "PASS",
     .arguments = REQUIRED,
     .handler = pass_handler},
    {.state = AUTHORIZATION,
     .name = "QUIT",
     .arguments = EMPTY,
     .handler = quit_handler},
    {.state = AUTHORIZATION,
     .name = "CAPA",
     .arguments = EMPTY,
     .handler = capa_handler},
    {.state = TRANSACTION,
     .name = "STAT",
     .arguments = EMPTY,
     .handler = stat_handler},
    {.state = TRANSACTION,
     .name = "LIST",
     .arguments = OPTIONAL,
     .handler = list_handler},
    {.state = TRANSACTION,
     .name = "RETR",
     .arguments = REQUIRED,
     .handler = retr_handler},
    {.state = TRANSACTION,
     .name = "DELE",
     .arguments = REQUIRED,
     .handler = dele_handler},
    {.state = TRANSACTION,
     .name = "NOOP",
     .arguments = EMPTY,
     .handler = noop_handler},
    {.state = TRANSACTION,
     .name = "RSET",
     .arguments = EMPTY,
     .handler = rset_handler},
    {.state = TRANSACTION,
     .name = "QUIT",
     .arguments = EMPTY,
     .handler = quit_handler},
    {.state = TRANSACTION,
```

```
    .name = "CAPA",
    .arguments = EMPTY,
    .handler = capa_handler}};
```

Limitaciones de la aplicación

Las limitaciones de nuestro servidor son las siguientes:

- La máxima cantidad de usuarios concurrentes es 512 y por lo que no tiene una gran disponibilidad para grandes volúmenes de conexiones.
- La cantidad máxima de conexiones pendientes para ser aceptadas es 24 por lo cual puede causar que algunas conexiones sean perdidas en caso de una gran carga de clientes intentan acceder al servidor en el lapso de tiempo muy pequeño.
- El servidor está implementado de manera bloqueante a la hora de leer archivos.
- Las configuraciones de `client` solo aceptan conexiones `ipv4`

Posibles extensiones

Las posibles extensiones que puede tener esta implementación del servidor y el protocolo usado son:

- Añadir los comandos opcionales indicados en el RFC 1939, como `TOP`.
- Métricas individuales para cada usuario.
- Modificar y ver el tamaño del buffer de lectura/escritura del servidor.
- Implementar un selector tal que convierta al servidor en no-bloqueante al leer archivos.
- Soporte de `ipv4` y `ipv6` en `client`

Respecto al protocolo desarrollado para la comunicación entre la aplicación cliente y servidor tenemos las siguientes consideraciones a mejorar:

- Aceptar otros mecanismos de autenticación para que las conexiones sean más seguras.
- Desarrollar códigos para las peticiones y sus respuestas, de forma que se pueda añadir al formato una zona numérica. Esto brindaría ayuda a entender si la solicitud fue exitosa, si hubo un error, o si se necesita alguna acción adicional.

Conclusiones

En conclusión, este trabajo práctico resultó ser un desafío integrador de los conceptos vistos a lo largo del cuatrimestre. Realizando esta tarea se logra percibir mejor y de manera más concreta (no tan abstracta) los temas vistos, a saber, funcionamiento de las capas de `aplicación` y `transporte`, la comunicación entre el `servidor` y sus `clientes`, manejo de `sockets`, entre muchos otros más.

En adición, nos permitió comprender de mejor manera las normas impuestas por los documentos `RFC` de manera práctica. Si bien estos archivos habían sido detallados en las clases teóricas, ver su aplicación y forma de estandarizar las conexiones nos demuestran lo útiles que resultan a la hora de realizar comunicaciones de este estilo.

Finalmente, en definitiva, creemos que la experiencia adquirida a través de este trabajo nos resultará beneficiosa en futuros proyectos que requieran conocimiento sobre protocolos de comunicación.

Ejemplos de prueba

Antes que nada se aclara que el servidor está inicializado con la siguiente línea de comandos para todos los ejemplos de prueba a continuación:

```
$$ ./pop3d --pop3-server-port 1110 --config-server-port 1120 -u someuser:pass123 -u someone:pass321 -d tmp/mails -v -t 123456
```

Pipelining

Corriendo el siguiente comando, podemos testear el `pipelining` correctamente.

```
$$ (printf "user someuser\r\npass pass123\r\nlist\r\n") | nc -C localhost 1110
```

Luego de ejecutar este comando, el servidor responde con lo siguiente.

```
+OK POP3 server ready
+OK valid user
+OK logged in
+OK
1 1783
2 3330
3 435
4 487
```

En este ejemplo se puede ver el correcto funcionamiento del `pipelining`, ya que se ve cómo el servidor puede recibir múltiples comandos sin tener que haber respondido a la instrucción anterior, luego procesa todos los comandos en el orden en el cual fueron enviados.

Cantidad de usuarios concurrentes

Se creó el siguiente script en bash, presente en el repositorio para su uso:

```
#!/bin/bash

iterations=512

cleanup() {
    echo "Cerrando conexiones..."
    for pid in "${pids[@]}"; do
        kill -TERM "$pid" 2>/dev/null
    done
    wait
    echo "Conexiones cerradas"
}

trap 'cleanup; exit 1' INT TERM EXIT

sleep_ms() {
    local duration=$(echo "$1 / 1000" | bc -l)
    sleep "$duration"
}

for ((i = 1; i <= iterations; i++)); do
    { printf "USER mdaneri\n"; sleep 10; printf "PASS pass123\n"; sleep 10; printf "QUIT\n"; sleep 10;} | nc -C localhost 1110 >> test_conc
    pid=$!
    pids+=("$pid")
    printf "Connection $i created with pid: $pid\n"
    sleep_ms 10
done

# Wait for 20 seconds
sleep 20

# Clean up and close connections
cleanup

echo "All iterations completed."
```

Este es un script de bash que corre los comandos `USER` `PASS` y `QUIT` en nuestro servidor luego de crear una conexión cada 10 ms (para poder dar tiempo al servidor a aceptar las conexiones).

```
USER someuser
PASS pass123
QUIT
```

Crea 512 conexiones de manera concurrente. Esto se puede ver en tanto el archivo test_concurrent_out.txt como en los logs del servidor.

En el archivo se pueden ver:

```
+OK POP3 server ready
+OK valid user
```

Impresos 512 veces seguidas, luego de esos 512 (que demuestra que se crearon las 512 conexiones), se puede ver 512 veces:

```
+OK logged in
```

Que demuestra que además de haberse creado las conexiones, estas no fueron abortadas en ningún momento y que se está atendiendo a los 512 clientes de forma concurrente.

Finalmente se ven 512:

```
+OK
```

Estos hacen referencia a los varios `QUIT` y al correcto cerrado de todas las conexiones.

En los logs:

```
[2023-24-11 15:25:02] [INFO]  Registering client with fd 7
...
...
[2023-24-11 15:25:09] [INFO]  Registering client with fd 518
[2023-24-11 15:25:29] [INFO]  Closing connection with fd 7
...
...
[2023-24-11 15:25:30] [INFO]  Closing connection with fd 518
```

También demuestra el hecho de que hayan estado concurrentemente las 512 conexiones.

Correcto funcionamiento al trabajar con un archivo de gran tamaño.

Prueba para una conexión

Generamos un archivo de 120KB utilizando el siguiente comando:

```
$$ dd if=/dev/urandom of=mail bs=120K count=1
$$ base64 mail > mail_base64
```

Luego, corrimos el siguiente comando para comprobar el correcto funcionamiento del servidor:

```
$$ nc -C localhost 1110 > output
$$ USER someuser
$$ PASS pass123
$$ RETR 1
```

```
$$ QUIT
$$ QUIT
```

Finalmente, para verificar que leimos correctamente del archivo, utilizamos el comando `diff` entre `mail_base64` y `output`, obteniendo:

```
$$ diff output ./New_maildir/someuser/cur/mail_base64 > diff_text
$$ cat diff_text
< +OK POP3 server ready
< +OK valid user
< +OK logged in
< +OK message follows
<
< .
< +OK
< +OK pop3 server signing off
```

Podemos apreciar las única diferencias son respecto de las salidas estándar del protocolo.

Guía de instalación detallada y precisa

Compilación y creación de los ejecutables

Para comenzar con la compilación, este proyecto cuenta con un Makefile que facilita este proceso. Para ello, entrar al directorio raíz y ejecutar los siguientes comando. El primero para limpiar los archivos residuales y el segundo para generar los ejecutables necesarios.

```
$$ make clean
$$ make all
```

Uso del directorio de mails

Para poder utilizar correctamente el servidor POP3, se requiere contar con un directorio que contenga todos los correos de los usuarios del sistema. A su vez, dentro de cada una de las carpetas de los usuarios, se debe agregar una carpeta llamada `cur` que incluirá los correos, cada uno en archivos separados. Este directorio debe especificarse con `-d` la dirección de este directorio.

A continuación, se adjunta un esquema que ilustra lo descrito previamente.

```
Maildir
└── someuser
    ├── cur
    │   ├── 1700513155.V802I61309M300068.debian_2,S
    │   └── 1700513186.V802I61e75M344960.debian_2,
    ├── new
    └── tmp
```

A continuación se presenta un ejemplo de uso:

```
$$ ./pop3d -u mdaneri:pass123 -u someone:pass321 -d ../Maildir -v -t 123456
```

Para conectarnos al servidor pop3d y utilizarlo.

```
$$ nc -C localhost [SERVER_PORT]
```

El flag `-C` es esencial para el funcionamiento, ya que envía el ASCII `'\r'` antes de enviar `'\n'`.

Instrucciones para la configuración

Antes que nada se debe iniciar el programa servidor, `pop3d`.

```
$$ ./pop3d --pop3-server-port [SERVER_PORT] --config-server-port [CONFIG_PORT] -u someuser:pass123 -u someone:pass321 -d [MAILDIR] -v -t [T
```

Los argumentos aceptados son los siguientes:

```
--help
-h                              This help message.
--directory <maildir>
-d <maildir>                    Path to directory where it'll find all users with their mails. (Optional)
--pop3-server-port <pop3 server port>
-p <pop3 server port>           Port for POP3 server connections.(Optional)
--config-server-port <configuration server port>
-P <configuration server port>  Port for configuration client connections.(Optional)
--user <user>:<password>
-u <user>:<password>            User and password for a user which can use the POP3 server. Up to 10.(Optional)
--token <token>
-t <token>                      Authentication token for the client.
--version
-v                              Prints version information.(Optional)
```

Para la ejecución se corre el siguiente comando:

```
$$ ./client [ARGUMENTS]
```

Ejemplos de configuración y monitoreo

Cada comando de configuración y monitoreo se realiza ejecutando el programa `client` al mismo tiempo que se brindan los argumentos necesarios.

Cada comando se envía de a uno al servidor y espera a obtener respuesta, o en caso de no recibirla, se avisa y se continúa con el siguiente comando.

Token de autenticación

Los comandos necesitan el token de autenticación, de 6 dígitos alfanuméricos, para validarse ante el servidor.

Tal token debe ser el primer argumento al iniciar la aplicación cliente, de la siguiente forma

```
$$ ./client -t 123456
```

Mensaje de ayuda

Ejecutando el siguiente comando en la terminal, obtenemos un mensaje con todos los comandos del servidor.

```
$$ ./client -h
```

Versión del protocolo

Ejecutando el siguiente comando en la terminal, obtenemos la versión del protocolo.

```
$$ ./client -v
```

El servidor responderá con el siguiente mensaje.

```
OK+ POP3 Server v1.0 | ITBA - 72.07 Protocolos de Comunicación 2023 2Q.
```

Añadir un usuario al servidor

Usando el siguiente comando, poniendo luego del flag `-t` el client y luego del `-u` el usuario y contraseña a agregar con el formato `<user:password>`

```
$$ ./client -t someuser -u newuser:newpass
```

Luego, el servidor responderá con el siguiente mensaje en caso de haberse creado exitosamente.

```
OK+ Added new user.
```

Cambiar la contraseña de un usuario

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego del `-p` el usuario junto a la nueva contraseña con el formato `<user:password>`

```
$$ ./client -t 123456 -p someuser:newpass
```

Luego, el servidor responderá con el siguiente mensaje en caso de haberse modificado exitosamente, o en caso contrario, mensaje de error.

```
OK+ Password changed.
```

```
-ERR. Can't change user password.
```

Eliminar un usuario del servidor

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego del `-r` el usuario a eliminar.

```
$$ ./client -t 123456 -r benedetto
```

Luego, el servidor responderá con el siguiente mensaje en caso de haberse eliminado exitosamente.

```
OK+ Removed user.
```

Cambiar el directorio de los mails

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego del `-m` el directorio a donde trasladar los archivos.

```
$$ ./client -t 123456 -m ../../../Maildir
```

Luego, el servidor responderá con el siguiente mensaje en caso de haberse trasladado exitosamente.

```
OK+ Maildir changed.
```

Establecer la cantidad máxima de mails

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego del `-s` la cantidad que queremos disponer.

```
$$ ./client -t 123456 -s 7
```

Luego, el servidor responderá con el siguiente mensaje en caso de haberse establecido exitosamente.

```
OK+ Max mails changed.
```

La cantidad máxima de mails cambiará para los usuarios que ingresen nuevamente desde la fase de autenticación. Los usuarios ya logueados, mantendrán la misma cantidad mientras su estado de transacción continúe.

Obtener la cantidad máxima de mails

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego agregar el flag `-g`

```
$$ ./client -t 123456 -g
```

Luego, el servidor responderá con el siguiente mensaje.

```
OK+ Max mails: 7.
```

Obtener la cantidad de conexiones históricas

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego agregar el flag `-i`

```
$$ ./client -t 123456 -i
```

Luego, el servidor responderá con el siguiente mensaje.

```
OK+ Historic connections: 20.
```

Obtener la cantidad de conexiones actuales

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego agregar el flag `-c`

```
$$ ./client -t 123456 -c
```

Luego, el servidor responderá con el siguiente mensaje.

```
OK+ Current connections: 5.
```

Obtener la cantidad de bytes transferidos

Usando el siguiente comando, poniendo luego del flag `-t` el admin y luego agregar el flag `-b`

```
$$ ./client -t 123456 -b
```

Luego, el servidor responderá con el siguiente mensaje.

```
OK+ Bytes transferred: 1024
```

Documento de diseño del proyecto

En el proyecto se pueden diferenciar dos partes: la parte del servidor y la parte del cliente ‘admin’.

En cuanto al servidor:

Para empezar en `main.c` se configuran los sockets de los clientes, los selectores y el logger.

Los archivos bajo el directorio stm, `pop3_stm.c` y `pop3_smt.h` se encargan en manejar la máquina de estados, procesar los comandos y el read, write y la autorización de estos.

El archivo `pop3_parser.c` y su header `pop3_parser.h` bajo el directorio parser se encarga de la transición de los estados del parser de los comandos de pop3.

El archivo `pop3.c` acepta las conexiones de los clientes nuevos y handlea la escritura y lectura de cada cliente dependiendo de su estado.

El archivo `args.c` bajo el directorio args posee las funciones y logica nescesaria para poder parsear y setear los argumentos recibidos por línea de comandos al inicializar el servidor.

El archivo `managment.c` en la carpeta managment contiene las funciones para poder aceptar las conexiones, argumentos y ejecutar los comandos del cliente admin.

En el archivo `server_constants.h` se encuentran gran parte de las estructuras, constantes y enums necesarios para el funcionamiento del servidor.

En cuanto al cliente:

En el archivo `admin.c` se encuentra toda la lógica para mantener comunicaciones con el cliente de monitoreo. Se configura el servidor y el cliente, para poder enviar por UDP los comandos que se ingresen.

Por otra parte, en `admin_args.c`, se encuentra el parseo de los argumentos ingresados para traducirlos a comandos así como si requieren argumentos o no para poder ser ejecutados.

Referencias

-Post Office Protocol - Version 3. J. Myers, Carniege Melon, M. Rose https://www.ietf.org/rfc/rfc1939.txt
-POP3 Extension Mechanism. C. Newman, R. Gellens, L. Lundblade https://www.ietf.org/rfc/rfc2449.txt