Trabajo Práctico Especial
Protocolos de Comunicación
(72.07)

2do Cuatrimestre 2018

Diseño y de desarrollo de un servidor proxy POP3

Integrantes:

Tomás Ferrer *57207*
Felipe Gorostiaga *57200*
Santiago Swinnen *57258*

Índice

Descripción detallada de aplicaciones y protocolos desarrollados

El protocolo diseñado para la comunicación entre el administrador y el filtro es de tipo binario. Teniendo en consideración el tipo de interacción entre el usuario (humano) y la consola, el ejecutable administrador transforma el texto ingresado en una estructura estática compatible con el protocolo que se describe a continuación y lo envía al proxy mediante un socket utilizando el protocolo de transporte SCTP.

El protocolo es de tipo request-response, con lo cual por cada solicitud se recibe una respuesta.

Protocolo

El protocolo presenta la siguiente estructura: cada datagrama tiene un número fijo de partes que presentan el mismo formato y cada una de ellas contiene la información necesaria para cada opción de configuración o métricas.

El **primer byte** de cada parte se utiliza para almacenar un número, que es el código ASCII de la letra designada para cada opción de la configuración. De esta forma, el proxy sabrá a qué variable de configuración asignar el parámetro que sigue. Las letras utilizadas para las cada opción se corresponden con aquellas utilizadas para pasar argumentos al proxy por línea de comandos al momento de la ejecución (aquellas que admiten modificación en tiempo de ejecución).

El **segundo byte** es un número de 0 a 255 que representa en largo en bytes del parámetro. Para el caso de que el tamaño sea mayor a 255 se toma el siguiente byte como un entero del mismo tipo que el anterior. Se repite el mismo procedimiento hasta que el siguiente byte sea menor a 255. Si la longitud fuera un múltiplo exacto de 255, entonces se utilizará el byte subsiguiente con contenido 0.

Los **bytes siguientes al tamaño** corresponden a los códigos ASCII de los caracteres que se ingresaron por parámetro (es decir el texto).

Aplicación cliente para administración del servidor

La aplicación administrador se encarga de leer comandos de entrada estándar ingresados por el usuario y luego enviarlos usando el protocolo mencionado anteriormente al servidor proxy y finalmente leer la respuesta y comunicar al usuario.

Primero se leen los argumentos del programa, los cuales tienen que ser la dirección IP y el puerto en el que se encuentra el servidor proxy, caso de que no se especifiquen, se procederá conectándose a localhost y puerto 9090. Una vez obtenidos estos datos se crea el socket SCTP y se lo conecta con el proxy.
Una vez creado y conectado exitosamente el socket, el usuario deberá ingresar la contraseña antes de poder ingresar a la consola de comandos. La contraseña se envía por el protocolo con el formato [x|size|password]. Habiendo ingresado correctamente la contraseña y leído un OK del servidor, se le otorga permiso al usuario a ingresar comandos para la administración del proxy server. Los comandos disponibles son los siguientes:

- M <mimes>, especifica los mimes a remover
- m <message>, cambia el mensaje de reemplazo
- t <comando>, cambia el comando a ejecutar a los mails recibidos
- z BYTES CON_CON TOT_CON, lee métricas del servidor por default -z pide todas
- f <filename> redirecciona métricas al archivo
- e <filename> redirecciona stderr al archivo
- q quit
- v muestra la versión del servidor
- h help

Funcionamiento en conjunto

Se agregan los siguientes ASCII como opcode:

x: clave para ejecutar el administrador.
z: métricas. Como parámetro se mandan las palabras identificatorias para cada métrica que se solicita:

- BYTES para obtener los bytes transferidos.
- CON_CON para obtener el número de conexiones concurrentes.
- TOT_CON para obtener el número de conexiones totales.

Ejemplo: se escribe por línea de comandos del administrador la siguiente entrada:

-m Mensaje con un largo superior a los doscientos cincuenta y cinco caracteres a modo de prueba. Mensaje con un largo superior a los doscientos cincuenta y cinco caracteres a modo de prueba. Mensaje con un largo superior a los doscientos cincuenta y cinco caracteres a modo de prueba.

El mensaje anterior tiene un largo de 280 bytes.
Entonces se manda el siguiente contenido al servidor:

‘m’ (1 byte)

255 (1 byte)
26 (1 byte)
Mensaje con un largo superior a los doscientos cincuenta y cinco caracteres a modo de prueba. Mensaje con un largo superior a los doscientos cincuenta y cinco caracteres a modo de prueba. Mensaje con un largo superior a los doscientos cincuenta y cinco caracteres a modo de prueba. (280 bytes)

La limitación a 255 bytes como máximo de contenido permite manejarse internamente con buffers de tamaño ilimitado. Para manejarse de forma transparente, el parser en la recepción espera a recibir un ASCII distinto de “c” para reconstruir el parámetro anterior y asignarlo a la estructura de configuración, o solicitar las métricas si este fuera el caso.

Respuesta del servidor

Las respuestas del servidor tienen el mismo formato que las solicitudes. Dentro del campo de contenido se envía desde el servidor texto con alguno de los siguientes formatos:

- Si la solicitud fue para modificar un parámetro, o para obtener autenticación:
    - OK. Si se modificó con éxito.
    - ERROR si falló la modificación.
- Si la solicitud fue para obtener métricas:
    - NOMBRE_MÉTRICA1 valor1 NOMBRE_MÉTRICA2 valor2.

Flujo general del servidor

Al ejecutar el filtro se inicializa una estructura con todos los parámetros de configuración con su valor default. Una variable `static` del tipo del TAD permite acceder a las funciones de modificación u obtención de valores de la configuración en cualquier momento de la ejecución del servidor de forma tal que la misma pueda sufrir modificaciones en tiempo de ejecución.

Para la resolución de nombres se utilizó la función `inet_pton`. Se consideró inicialmente darle un tratamiento especial a la resolución utilizando un nuevo hilo de ejecución pero como en la práctica no se presentaron inconvenientes ni demoras de ningún, el mismo se ejecuta en el hilo principal.

En primera instancia se leen los valores de configuración pasados como argumentos por línea de comando y a continuación se crean dos sockets pasivos: uno para establecer una comunicación TCP para POP3 y otro para comunicarse con SCTP con administradores remotos utilizando el protocolo descrito en la sección anterior.

Una vez completados los pasos iniciales para que se establezca correctamente el servidor, el proxy entra en un loop infinito en el que lee los sockets pasivos para ver si hay nuevos pedidos de conexiones y crea nuevos sockets para comunicarse con los clientes. Se verifica además, si los clientes ya conectados con el servidor enviaron contenidos por el socket. En caso de hallar contenido para leer, se utilizan distintos parsers según la proveniencia del socket. Para conocer este dato, el servidor mantiene un array paralelo de enteros, donde cada posición tiene un número que corresponde a la forma de comunicación que se debe utilizar. Por ejemplo, si en la posición 3 del array de sockets se encuentra el descriptor de un socket que se comunica con el administrador, en la misma posición del array paralelo se encontrará el valor correspondiente a la constante `ADMIN`. También se utilizan arrays paralelos para los sockets que se van creando al origin server y para los pipes de entrada y salida al filtro. De esta forma, para todos los arrays mencionados todas las posiciones i de los mismos corresponden a los pipes y sockets necesarios para el funcionamiento de cada cliente POP3 diferente.

Resulta esencial para el correcto funcionamiento de un servidor que no se produzcan bloqueos durante su ejecución. Es por este motivo que se utilizan las función `select` para detectar cuando hay actividad en los sockets y pipes y con `fcntl` se los para que sean no bloqueantes. De esta forma queda garantizado que no habrá bloqueos durante la ejecución. Además, se debe respetar una forma de lectura y escritura de los sockets para que no se entorpezca el uso de los sockets no bloqueantes. Con este fin se utiliza el valor `EWOULDBLOCK` de `errno` para saber en qué momento dejar de hacer lecturas al socket del marcado por el selector, dado que se contempla la posibilidad de que el buffer sea de tamaño muy pequeño para la escritura que realiza el cliente o que éste último realice la escritura de forma fragmentada. En la sección siguiente se describe con mayor detalle la lógica aplicada a éste aspecto. No se utilizó ningún parche provisto por la cátedra bajo el supuesto de que resulta un valor agregado resolver la arquitectura únicamente mediante el uso de funciones de librerías permitidas.

Algo similar sucede con el logging. Si bien una escritura a la terminal funcionaría sin problemas, en el caso de que la salida estándar se redirija y sea necesario escribir a un pipe o a un archivo en disco, esto sí podría generar bloqueos en el proxy. La decisión tomada es únicamente marcar la salida estándar como no bloqueante. Con las escrituras a los sockets y pipe no se utiliza un selector, dado que el flujo alterna lecturas y escrituras de tamaño relativamente pequeño, quedando una baja probabilidad de eventual bloqueo. De todas formas, al estar marcados los file descriptors como no bloqueantes, esto jamás produciría un bloqueo sino que se toma este comportamiento a riesgo de que alguna escritura no se realice completamente, lo cual no sucedió durante las pruebas realizadas.

Para el caso en que se detecte contenido en un socket destinado a la administración, se utiliza un parser específicamente diseñado para recibir mensajes del protocolo diseñado para este tipo de interacciones. Tanto el parser como sus utilidades auxiliares se encuentran en el archivo `configParser.c`. Este parser lee el contenido del socket e interactúa con la estructura de configuración para obtener o modificar su contenido según sea necesario.

Para el caso en que se detecte contenido en un socket proveniente de un cliente POP3, se utiliza un parser cuya implementación se encuentra en el archivo `pop3parser.c`. Se detallará su funcionamiento en una sección aparte a continuación.

Atención a clientes POP3

Se identifica un flujo entre 3 extremos y el proxy como pivote.

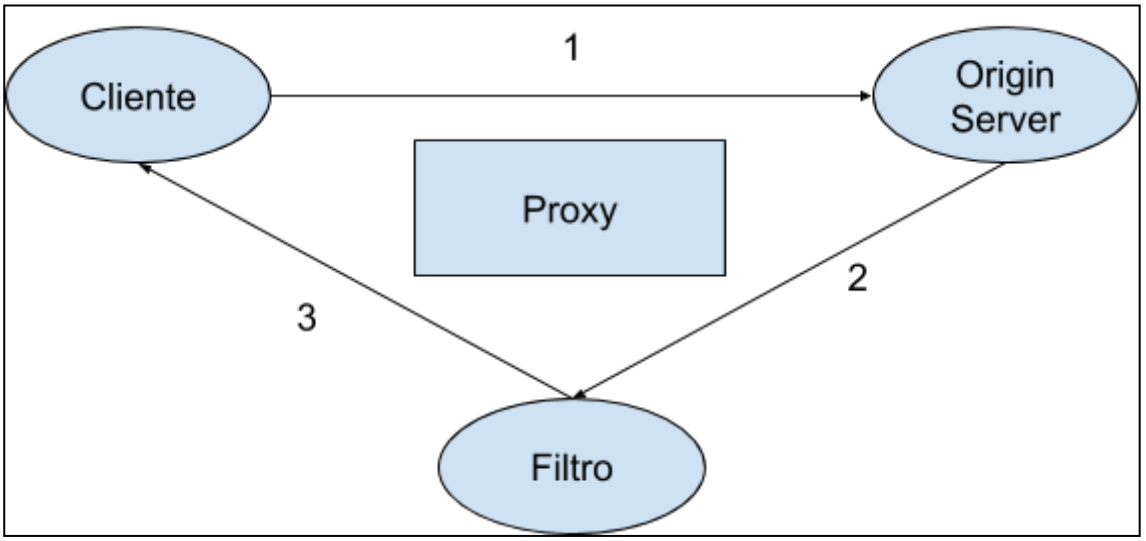

Gráfico 1

Para la implementación de la atención a los clientes se partió de las siguientes premisas:

- Nunca se debe esperar nada de ninguna parte, ya que ante cualquier error, la consecuencia sería un bloqueo, lo cual en un servidor es una consecuencia grave. Es por este motivo que no se realizan llamadas a read sino mediante el selector.
- El orden de estos pasos en el selector (si hubiera necesidad de lectura en los tres casos) primero el 1, luego el 2, después el 3. Esto no quiere decir que si estuvieran en otro orden no funcionará, sino que se elige este orden por dos cuestiones:
    - Facilita la comprensión de la lógica de la progresión del código.
    - Facilita el cierre de pipes y sockets en caso de que esto fuera necesario.

Filtrado de mensajes de correo electrónico

La aplicación de filtrado de mensajes de correo electrónico remueve las partes de los correos electrónicos originales que recibe de la aplicación pop3filter, usando como criterio para la eliminación los tipos MIME que recibe también de la misma aplicación. Las principales fuentes para asegurar el funcionamiento del parseo de los mensajes fueron los documentos RFC 2045 (Multipurpose Internet Mail Extensions (MIME) Part One), 5322 (Internet Message Format), 822 (STANDARD FOR THE FORMAT OF ARPA INTERNET

TEXT MESSAGES), 1341 (MIME (Multipurpose Internet Mail Extensions) y 7231 (HTTP, por la definición de MIME types para el header Accept).

Para el desarrollo de esta aplicación fue orientativo el ejemplo visto en clase de la pequeña aplicación que detecta tipos MIME y el header Content-Type.

La aplicación comienza inicializando estructuras y campos de diversos tipos, así como recibiendo y separando la lista de tipos MIME. Una vez terminadas estas tareas, se comienza a parsear el mensaje, buscando primero posibles respuestas de comandos POP3 que podrían ser enviadas a la aplicación. Una vez hecho esto comienza la revisión de los Headers principales del mensaje. En particular se buscan Headers denominados “relevantes”, que pueden interferir con el proceso de censura. Esto se debe a que la remoción de una parte del mensaje no implica la remoción completa de esta, sino el reemplazo de su Body por uno especificado por pop3filterctl y obtenido de la variable de entorno FILTER_MSG.

Todos los headers se van guardando de a uno en un buffer, y si se determina que el header no es relevante se envía directamente a salida estándar (cabe destacar que no se determina su relevancia una vez que se obtiene por completo; se va buscando letra por letra hasta que se encuentre una que no coincida). Si el header, no obstante, es un header relevante, este se guarda en un arreglo temporalmente hasta que se sepa si la parte en cuestión debe o no ser censurada. Los headers relevantes son:

- Content-Transfer-Encoding: Este es el de mayor importancia (luego de Content-Type por supuesto), ya que las partes a filtrar pueden tener un encoding completamente distinto a us-ascii (el tipo por defecto) y pueden provocar que se muestre erróneamente el mensaje censurado, ya que será considerado encriptado por más que no lo sea.
- Content-MD5: Algún programa podría usar este campo para determinar la validez de la parte entrante.
- Content-Length: Puede causar lecturas de tamaño indebido.
- Content-Type: Este es quien determina si los headers anteriores deben ser incluídos o no. Si el tipo MIME presente en este header coincide con uno perteneciente a la lista obtenida inicialmente (no debe necesariamente coincidir del todo, también hay comodines MIME denotados con un *). Si coincide es reemplazado por Content-Type: text/plain, ya que en esta parte del mensaje se insertará solamente un mensaje de reemplazo.
- Se evaluó también la posibilidad de incluir el header Content-Disposition, ya que al filtrar una parte podría suceder que el correo contenga un archivo con cierta extensión que no permitiera que se abra correctamente. Sin embargo, quitarlo resultaría en que el correo contenga una o más partes de texto sin ninguna referencia a los archivos perdidos. Esto podría haberse evitado parseando el atributo filename de este header, pero no fue considerado grave. Dejando este header intacto, en cambio, el usuario sabe exactamente qué archivos le causan problemas.

Cabe destacar también que los correos pueden estar compuestos por diferentes partes por lo que no tendría sentido parsear solamente los headers iniciales. Es por ello que además de comparar el cuerpo del campo Content-Type contra los MIMEs relevantes censurables, se los compara con los tipos MIME compuestos (en particular, los de la forma multipart/* y los message/rfc822). No se consideró necesario censurar otros subtipos de message ya que podrían regirse por otras normas ajenas al formato de correo electrónico. Si el mensaje es multipart entonces se parsea su atributo boundary (obligatorio para los tipos multipart según RFC), el cual delimita las distintas partes del mensaje. Si el mensaje es message/rfc822 se sabe que el cuerpo del mensaje será en realidad otro correo, por lo que se debe seguir parseando headers del mismo formato del correo actual.

Entonces, si el mensaje es multipart, se revisará cada parte incluida en estas partes compuestas. Se toma en cuenta la posibilidad de partes multipart y/o message/rfc822 recursivas. Para esto fue necesaria la implementación de una pila en la cual ir apilando los boundaries. Se sabe que nunca puede terminar un multipart introducido después antes de que terminen todos los multipart de niveles superiores. Los boundaries, asimismo, deben estar correctamente anidados.

Si se filtrara un MIME compuesto simplemente se filtrarían sin ningún tipo de revisión todas las partes incluidas por una parte de este tipo compuesto.

Se contempla la posibilidad de “Line folding”, así como la presencia de comentarios en los headers, tal y como definidos en los documentos RFC mencionados anteriormente (en particular el 5322). También se tiene en cuenta que los headers son case-insensitive.
En el proceso de filtrado podrían eliminarse algunos espacios irrelevantes en los headers, así como ciertos comentarios ubicados en algunos headers relevantes.
Podría alterarse el orden de los headers (lo cual no cambia semánticamente al mensaje) ya que los headers relevantes siempre son enviados a salida estándar una vez que se determina si la parte debe ser filtrada o no.

Una vez finalizado el parseo de headers, se pasa directamente al parseo del cuerpo del mensaje. Aquí, el parser de headers se habrá encargado de determinar si el cuerpo debe ser censurado o no, por lo que se enviarán los bytes a salida estándar si no debía ser filtrado, y si debía ser filtrado, se insertará el cuerpo indicando que la parte fue filtrada. Para el cuerpo del mensaje no se utiliza ningún buffer auxiliar: lo que se lee se envía de a cierta cantidad de caracteres.

El parser del cuerpo de los mensajes cumple además la función de encontrar los límites de las partes del correo. Es aquí cuando se desapila de la pila cualquier boundary de partes que hayan terminado. Una vez que se termina de parsear el cuerpo del mensaje, se vuelve a parsear Headers si aún quedan partes sin leer, y si el correo termina, entonces se liberan recursos y se termina la ejecución.

El programa va leyendo (y escribiendo) bytes de a *chunks* de cierta cantidad de bytes determinada en tiempo de compilación (a excepción de los headers, que son enviados de a uno por vez por salida estándar una vez que se hayan terminado de parsear).

Probando el caso de prueba de un correo con un archivo adjunto de gran tamaño (el correo tiene un tamaño de 664.5 MB), se probaron tres distintos tamaños de buffer, y los resultados fueron los siguientes:

Tamaño de Buffer	Tiempo: real	Tiempo: user	Tiempo: sys
1	(aprox. 25 MB transferidos en 1 min, ~405KiB/s)	-	-
100	0m29.618s, ~22,4MiB/s	0m14.199s	0m44.342s
1000	0m15.998s, ~66MiB/s	0m8.806s	0m8.636s

Tabla: tiempos obtenidos con el comando:

```
time cat i_big.mbox | ./stripmime | pv > out_big
```

PC usada: Lenovo Flex 5, Intel Core i5-8250U, Ubuntu 16.04.3 instalado en HDD 5400RPM

Problemas encontrados durante el diseño y la implementación

Sobre el diseño del protocolo

En una primera instancia se había considerado un protocolo en el que cada datagrama tuviera un tamaño limitado de 1 byte de tamaño, 1 para el opcode y 255 como máximo para el contenido. En caso de que fuera necesario, se marcaría el siguiente datagrama con un código específicamente reservado para indicar que el mensaje continuaba. Sin embargo, esto presentó el siguiente inconveniente: resultaba imposible saber desde el lado del receptor si el mensaje continuaba o no, a menos que se analizara el byte de tamaño.

Si bien se consideró analizar el byte de tamaño y en caso de que este fuera 255 esperar otro datagrama más, en los casos en que la longitud fuera múltiplo de 255 se enviaría un datagrama vacío, lo que representaba realizar una escritura extra al socket y su correspondiente lectura innecesariamente, cayendo en una utilización subóptima de los mismos, por lo menos desde lo conceptual (aunque es probable que las consecuencias en la práctica fueran mínimas).

Sobre la interacción con el cliente, el filtro y el servidor de origen

En lo que posiblemente fue la determinación central de la implementación del servidor, se resolvió disponer el flujo entre cliente, servidor y filtro de la forma descrita anteriormente. Antes de llegar a esta solución se analizaron las siguientes cuestiones:

- La idea original sobre el tratamiento de los comandos recibidos del cliente era mantenerse parseando las respuestas del origin server y de los filtros antes de volver al selector. Sin embargo, a falta de poco tiempo para la fecha de entrega, se encontró que había ciertas partes que resultaban bloqueantes, o que potencialmente podía resultarlo. Se tomó la decisión de migrar a una forma completamente no bloqueante, pagado las consecuencias en otros aspectos, como por ejemplo la falta de soporte para origin servers no bloqueantes, que originalmente funcionaba, pero en esta entrega final no. Se priorizó cumplir con la premisa conceptual básica del trabajo que es no bloquearse en ningún momento bajo ninguna circunstancia.

Limitaciones de la aplicación

A continuación se detallan las limitaciones de la aplicación presentada:

- **Caída y recomienzo del servidor origen:** en caso de que caiga el servidor origen (caso que está contemplado en la implementación) se cierran los sockets a los clientes anunciando que se ha roto la conexión con el servidor. Si bien el proxy sigue ejecutando y recibiendo nuevas conexiones tal como se indica en el caso de prueba, se le envía a los nuevos clientes un mensaje indicando que no se pudo conectar.
  La mejor opción sería que cuando el servidor origen vuelve a estar disponible, las nuevas conexiones puedan alcanzarlo. Se propone una solución a éste caso en las posibles extensiones, pero al no ser una cuestión central del trabajo y tampoco representa un gran aporte su implementación, la decisión es simplemente informar a las conexiones entrantes que el servidor no está disponible.

- **No se soporta conexión a servidores sin pipelining**: como se comentó anteriormente esta funcionalidad no está presente. De todas formas se indica correctamente al usuario dicha situación y se cierran los pipes y sockets para que el proxy siga recibiendo conexiones e informando a los clientes nuevos el problema.

- **La prueba de los trailing spaces falla** : la escritura del proxy al origin server es completamente transparente, y conectándose directamente al puerto 110 desde la terminal, los trailing spaces también se producen. Se priorizó mantener la transparencia en el envío de comandos, si bien esta falla no es deseable.

- **Fallo en caso específico de censura:** hay un caso muy específico en el cual stripmime censura a pesar de que no debería hacerlo: en caso de que el MIME type a censurar sea, por ejemplo, application/x-mime-1, y se encuentra el header:
  Content-Type: application/x-mime-1

-sorpresa

Entonces se censurará indebidamente el MIME type. Cabe destacar que esto sucederá si y sólo si hay line folding (como definido en RFC 5322) y si justo se corta al MIME type de tal forma que lo que aparezca en la línea superior sea exactamente igual al MIME type que se desea censurar. Ninguno de los siguientes headers presenta problemas:

Content-Type: application/x-mim
e-1-sorpresa
Content-Type: application/x-mime

Posibles extensiones

- **Caída y recomienzo del servidor origen:** en base a la limitación comentada en la sección anterior se propone la siguiente extensión. Utilizando una variable de la estructura de configuración que indica si la conexión con el el servidor está activa o no, se podría verificar, luego de una determinada cantidad de ciclos del selector o un intervalo de tiempo, verificar si el servidor está disponible nuevamente. En caso de que esto suceda, la variable volvería a indicar que el servidor está activo. De esta forma se evitaría que durante la inactividad cada nueva conexión intente realizar una conexión fallida (con solo chequear la variable bastaría), y además el proxy volvería a ser de utilidad cuando se reanuda la ejecución del servidor origen.
- **Aumentar la capacidad del administrador:** las posibilidades del administrador hasta el momento habilitan un conjunto sumamente acotado de acciones. En este marco se le podría otorgar la posibilidad de modificar la dirección del servidor origen y las direcciones y puertos donde escucha cada socket pasivo.
- **Soportar servidores sin pipelining:** como se describió en la sección de limitaciones, esta funcionalidad no es provista.

Conclusiones

La realización del trabajo presentó dos grandes desafíos desde el punto de vista del diseño y desde el punto de vista de la implementación. Estos dos fueron el programa externo para el procesamiento del contenido del mail (stripmime), y la triangulación entre cliente, servidor y filtro mediante conexiones no bloqueantes

También resultó enriquecedor el proceso de diseño de un protocolo, elemento central de la materia, con todos sus problemas y respectivas soluciones.

Una vez obtenida una aplicación que resuelva las situaciones básicas que el alcance del trabajo requiere solucionar, resulta casi automático empezar a visibilizar posibles alternativas tanto de implementación como de diseño. Si bien este tipo de aplicaciones siempre son perfectibles, el diseño, el desarrollo, las decisiones y el resultado obtenido permite obtener un panorama más amplio sobre desarrollo de servidores proxy en general,

ya que la mayoría de las características de la aplicación no son necesariamente inherentes al protocolo POP3.

Ejemplos de prueba

Se siguió la documentación del archivo pop3.filter.8 provisto por la cátedra para los argumentos de ejecución, sin ningún tipo de excepción, incluyendo la convención de nombres, motivo por el cual únicamente se expondrá un caso de forma demostrativa ya que su ejecución es prácticamente idéntica a la detallada en el enunciado.

Corriendo `./pop3filter localhost` en una terminal y `nc localhost 1110` en otra, se pueden mandar comandos POP3 al origin server mediante el proxy que escucha en el puerto TCP 1110.

Guía de instalación e instrucciones de configuración

Posicionado en la carpeta principal del repositorio, ejecutar el comando `make clean`, seguido de `make`. Esto compilará todo el código y generará los ejecutables en la misma carpeta. Luego ya se pueden ejecutar el proxy y el administrador con los siguientes nombres:

- pop3filterctl para el administrador. Puede correrse sin argumentos o con la IP (de cualquier tipo) , seguida del puerto al que hay que conectarse. Si se corre sin argumentos la conexión se realiza a localhost en el puerto default del proxy. Cualquier otro formato de argumentos se ignorará. Un ejemplo:

  ```
  ./pop3filterctl  127.0.0.1   1256
  ```

- pop3filter para el proxy. Se utiliza la convención de argumentos provista por la cátedra sin ningún tipo de modificación.

Ejemplos de configuración y monitoreo

Correr el servidor:

```
./pop3filter localhost
```

Correr el administrador:

```
./pop3filterctl
```

Inicialmente se pide la password que es *adminpass*. Luego ya se pueden ejecutar comandos. Por ejemplo:

Este comando imprimirá el total de bytes transferidos y las conexiones concurrentes:

```
-z BYTES CON_CON
```

El siguiente cambia el comando de ejecución del filtro al stripmime propio:

```
-t ./stripmime
```

Documentación del diseño de la aplicación

El esqueleto de la aplicación, es decir, la inicialización del proxy y el selector, se encuentran en el archivo proxy.c. Se mantiene la información necesaria en un TAD de configuración definido en configuration.h al que se accede desde proxy.c únicamente.

Se separó lo relativo al administrador de lo relativo a POP3 en dos archivos, configParser.c y pop3parser.c, respectivamente. Éstos contienen las funciones que son llamadas al momento de leer y realizan las acciones necesarias según el caso, siempre y cuando éstas no involucren nuevas lecturas, que se harán una vez que se retorne al selector.

En la carpeta Admin se encuentra el código necesario para ejecutar el administrador y el código del stripmime está contenido dentro de la carpeta Stripmime. Ambos tienen un único archivo de extensión .c con su correspondiente .h

Se creyó conveniente ir describiendo detalladamente la arquitectura a lo largo de las distintas secciones, motivo por el cual esta última sección se presenta a modo de resumen de lo descrito anteriormente.