PROXY HTTP
Protocolos De Comunicación
1er Cuatrimestre 2018

GRUPO 8

Daniella Liberman - 57548
Giuliano Scaglioni - 57244
Marco Fallone - 48179
Matías Ota - 55099

1. Protocolo

Para definir el protocolo de administración/configuración entre el proxy y el cliente de administración, tuvimos en cuenta cuáles eran nuestros requerimientos, llegando a los siguientes:

Las tareas que necesitamos son:

- Autenticar (enviar credenciales)
- Listar métricas
- Listar configuraciones
- Pedir métricas
- Pedir configuraciones
- Setear configuraciones

Con estos requerimientos, definimos entonces, un protocolo binario, orientado a conexión, de tipo *request/response*, donde en la request se envía la tarea que se quiere realizar y en la response el resultado de ésta. Por tratarse de un protocolo binario, definimos también las correspondientes funciones de serialización.

Estructura de un mensaje

- **type**: 1 Byte que representa la acción.
    - 0: Enviar credenciales (pass-phrase)
    - 1: Listar métricas
    - 2: Listar configuraciones
    - 3: Pedir métricas
    - 4: Pedir configuraciones
    - 5: Setear configuraciones
- **param**: 1 byte que indica (de ser necesario) de qué métrica/configuración se trata. El número representado por el byte será el número de métrica/configuración de sus respectivas listas.
- **buffer_size**: 4 bytes que indican (de existir un mensaje) el tamaño del mensaje que sigue.
- **buffer**: mensaje del tamaño de message_size.

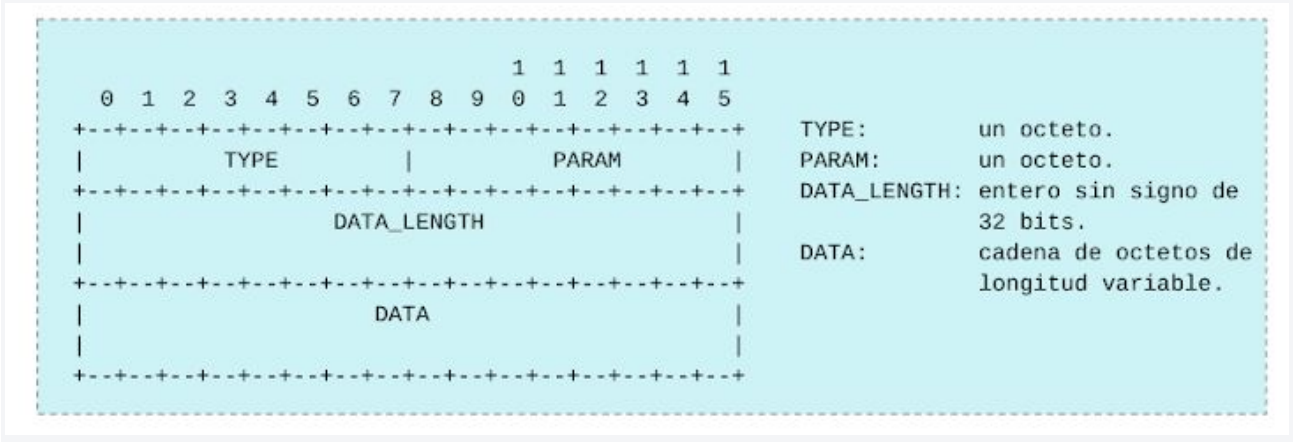
Los mensajes serán:

- Enviar credenciales
- La lista de métricas
- La lista de configuraciones
- El valor de una métrica
- El valor de una configuración
- Indicar error

Definición de constantes

```
SEND_CREDENTIALS  0
LIST_METRICS      1
LIST_CONFIGS      2
GET_METRIC        3
GET_CONFIG        4
SET_CONFIG        5
ERROR             6
```

Campos de un mensaje



El único campos que siempre estará inicializados será *type*. A éste se les agregan los demás dependiendo del tipo de mensaje:

- **SEND_CREDENTIALS**: Si se está del lado del administrador: *buffer_size* y *buffer*.
- **LIST_METRCIS y LIST_CONFIGS**: Si se está del lado del proxy: *buffer_size* y *buffer*.
- **GET_METRIC y GET_CONFIG**: *param* y si se está del lado del proxy: *buffer_size* y *buffer*.
- **SET_CONFIG**: Si se está del lado del administrador: *param*, *buffer_size* y *buffer*.

Por otro lado definimos también un protocolo de errores. Este cuenta con la definición de los siguientes tipos de errores

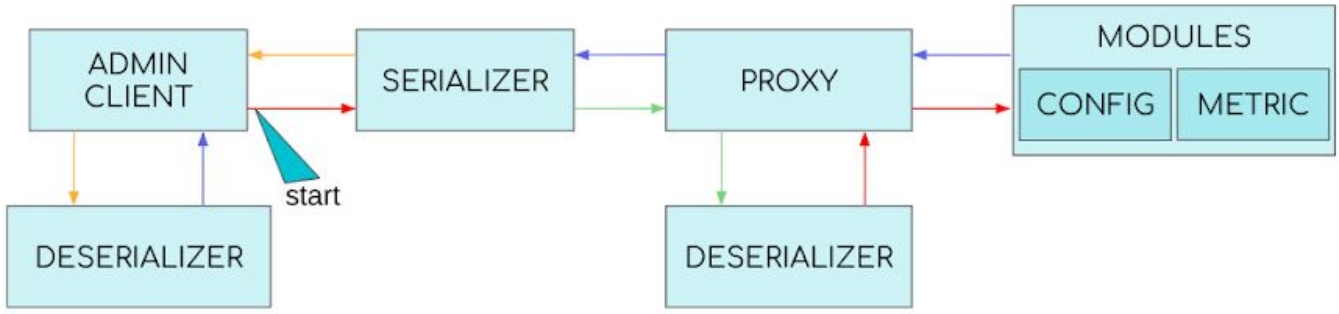
```
CONFIG_NOT_FOUND  0
METRIC_NOT_FOUND  1
CONFIG_NOT_SET    2
INVALID_LENGTH    3
UNEXPECTED_ERROR  4
```

En cuanto al contenido de buffer, siempre es un string de caracteres visibles (isgraphic) los cuales permiten ser directamente impresos. La representación del entero (DATA_LENGTH) es little endian. Es por esto que durante la serialización se realiza la conversión entre esta convención y la de la máquina en que corre el programa.

Breve descripción de la implementación

Contamos con una cola de mensajes (módulo msg_queue) a enviar al cliente administrador. De esta forma, cada vez que el selector indica que es posible escribir en el socket del cliente administrador, envía el próximo mensaje de la cola. Creemos que fue interesante la posibilidad de usar un socket del tipo one-to-many que ofrece la implementación del protocolo SCTP, de esta forma no se mantiene un socket por cada cliente (consumiendo file descriptors) sino que se mantiene un único socket con múltiples asociaciones.

El envío de un mensaje puede representarse resumido en el siguiente gráfico.



En C, un mensaje está representado por una estructura msg_t:

```
typedef struct {
   unsigned char   type;
   unsigned char   param;
   unsigned int    buffer_size;
   unsigned char * buffer;
} msg_t;
```

Mejoras

Por cuestiones de tiempo, no llegamos a implementar la funcionalidad de enviar credenciales. Como discutimos en clase durante la presentación, hubiera sido deseable tener un campo con la versión del protocolo. Esto último nos hubiera permitido actualizar el protocolo manteniendo retrocompatibilidad con clientes con versiones más viejas.

2. Descripción de módulos

En esta sección se describen en detalle los módulos más importantes (dejando de lado aquellos provistos por la cátedra) que conforman el trabajo. Para un detalle más breve sobre estos y cómo se relacionan entre sí, referirse a la sección *Arquitectura de la aplicación* .

2.1 Modulo Métricas

Permite calcular métricas en base a eventos producidos en el servidor. Las métricas que soportamos son:

- Cantidad actual de conexiones concurrentes
- Máximo histórico de conexiones concurrentes
- Cantidad de accesos
- Cantidad de bytes transferidos (client->proxy + proxy->client)

Las métricas en C se representan por un número de la siguiente manera:

```
enum metrics {
  INST_CONCURRENT_CONNECTIONS=0,
  MAX_CONCURRENT_CONNECTIONS,
  ACCESSES,
  TRANSFERED_BYTES,
  AVG_CONNECTION_TIME
}
```

Se cuenta también con la función metric_get_name que recibe un número de métrica y devuelve el nombre que la representa. Esta se usa cuando se envía el mensaje al cliente administrador. El valor de las métricas es de tipo *double*, y se cuenta con una función metric_get_value_stringque devuelve un string con el value. Esta se usa cuando se envía el mensaje al cliente administrador. El código de error para cuando una métrica no es encontrada es el -1, ya que las métricas son siempre cantidades o tiempos y no pueden ser negativas.

- **Conexiones concurrentes:** Lleva registro la cantidad de clientes que hay conectados (en ese instante) al proxy. Para esto, cada vez que un nuevo cliente se conecta, se llama a la función new_connection la cual incrementa en uno el valor de la métrica *INST_CONCURRENT_CONNECTIONS*. A su vez, cuando un cliente finaliza su conexión, se llama a la función end_connection que decrementa en uno el mismo valor.
- **Máximo de conexiones concurrentes:** Lleva registro de la máxima cantidad de conexiones concurrentes que hubo hasta el momento. Para esto, cada vez que un nuevo cliente se conecta, en la función new_connection mencionada anteriormente, se compara el valor resultante de *INST_CONCURRENT_CONNECTIONS* con el valor de *MAX_CONCURRENT_CONNECTIONS*. Si el primero supera al segundo, entonces se actualiza el nuevo máximo.
- **Cantidad de accesos históricos:** Lleva registro de la cantidad de clientes que se tuvo sin contar los que aún están siendo atendidos. Para esto, en cada llamada a la función end_connection aumenta en uno el valor de la métrica *ACCESSES*.

- **Bytes transferidos:** Lleva registro de la cantidad de bytes que fueron transferidos por el proxy, tanto hacia el client como hacia el origin. Para esto, luego de la transferencia, se llama a la función add_transfered_bytes la cual suma al valor *TRANSFERED_BYTES* la cantidad de bytes transferidos.
- **Avg connection time:** Lleva registro del promedio de tiempo en que un cliente se mantiene conectado al proxy. Para esto, en cada llamada a new_connection se crea una estructura de tipo *connection_metrics_t* y se almacena el tiempo inicial de ese cliente. Luego, en cada llamada a end_connection se pasa por parámetro esta estructura y se calcula con el tiempo actual y el tiempo almacenado en la estructura, cuánto tiempo estuvo ese cliente conectado. Luego, utilizando la variable estatica *connection_time_sum* (que lleva registro de la suma de los tiempos de atención a cada cliente), se almacena en *AVG_CONNECTION_TIME* el valor de *connection_time* / *ACCESSES* (luego de sumarle a *ACCESSES* el cliente actual).

2.2 Modulo Config

Este módulo se encarga de almacenar configuraciones. Creímos útil agregar este módulo ya que permite que se puedan centralizar las configuraciones.

La forma de disponibilizar los cambios en ellas en tiempo de ejecución es que cada cliente, cuando crea los buffers o inicia su programa de transformaciones, las obtiene. Esto hace que un cambio en una configuración se vea reflejada en el próximo cliente que se conecte.

- **struct config**: Tiene dos campos: *name* es el id de la configuración; *value* el valor que esta configuración tiene seteado. El valor es de tipo char * dado que puede ser de este tipo o int, en cuyo caso se utiliza la función atoi.
- **configurations**: Tiene todas las configuraciones que fueron seteadas.
- **MAX_CONFIG**: Se soporta hasta un máximo de MAX_CONFIG por lo que se cuenta con un array *configurations* de este tamaño.
- **MAX_NAME**: Se soporta hasta un máximo de caracteres en el configuration name MAX_NAME. En caso de que pasen un name con mayor tamaño, si coinciden en MAX_NAME, tomara que se trata de este mismo.
- **NAME**: Acepta letras, números, '_', '-'

El módulo también incluye la posibilidad de cargar configuraciones de un archivo, pero esta funcionalidad no fue utilizada.

2.3 Modulo Client

El módulo client se encarga de atender a los clientes. Internamente funciona como una máquina de estados. Requiere que se le informe de los eventos del selector. Estos eventos ejecutan distintas acciones dependiendo del estado actual del cliente. Para esto, el módulo expone cuatro handlers para cada uno de los posibles eventos sobre el file descriptor del socket cliente. Internamente también se usan otros file descriptors para el socket utilizado en la conexión con el origin y las transformaciones. La máquina de estados también evoluciona frente a eventos en estos file descriptors, pero en el contrato no se exponen los handlers pues el registro de los file descriptors en el selector lo hace el módulo internamente.

Estados

Los estados que maneja internamente el módulo client en su máquina de estados son:

- **NO_ORIGIN**: Cuando aún no se realizó la conexión al origin.
- **SEND_REQ**: Cuando se está escribiendo la request en el origin.
- **READ_RESP**: Cuando se está leyendo la respuesta desde el origin.
- **ERROR**: Cuando se produjo un error.

Además se cuenta con dos flags que indican si ya se leyó completamente la request/response actual, estos son: *request_complete* y *response_complete*. Estos flags deberían ser seteados por el parser.

Acciones ante eventos

Los siguientes son los posibles eventos que pueden llegar desde el selector y las acciones que toma la máquina de estados dependiendo del estado actual.

Aclaración: cuando se pasa la request por request parser, este debería dejar los caracteres consumidos en *post_req_parse_buffer*. Cuando se pasa la response por el response parser, este debería dejar los caracteres consumidos en *post_res_parse_buffer* siempre que no haya transformación a realizar. En caso de transformación, se pasan los headers a *post_res_parse_buffer* y se deja en *pre_trans_buffer* el body de la request.

Leer del cliente

- **NO_ORIGIN**: Siempre que haya lugar en el buffer, pasamos la request por el request parser. Cuando este obtenga el host, deberia llamar al callback para setear el host, iniciandose la resolucion del nombre y posteriormente, la conexion.
- **SEND_REQ**: Pasamos la request por el request parser.

Escribir hacia el cliente

- **READ_RES**: Si hay caracteres en *post_res_parse_buffer*, los enviamos al cliente. Si no hay caracteres en *post_res_parse_buffer* y está seteado *response_complete* pasamos al estado READ_REQ (reiniciando el estado de los flags).
- **NO_REMOTE**: Si hay caracteres en el buffer de salida, los enviamos al cliente. Si no hay caracteres en el buffer de salida liberamos los recursos y hacemos close del cliente.
- **ERROR**: Chequeamos que error se produjo, y si es necesario, se envía al cliente una response acorde.

Desbloqueo en el cliente

- **NO_ORIGIN**: Hacemos connect al origin y pasamos al estado **SEND_REQ**.

Close en el cliente

Liberamos los recursos del cliente (buffers, parsers, socket al origin, fd's, etc.).

Leer en origin

- **READ_RESP**: Pasamos la response por el response parser.

Escribir en origin

- **SEND_REQ**: Si hay caracteres en *post_req_parse_buffer*, los enviamos a origin. Si no hay caracteres en *post_req_parse_buffer* y esta seteado *request_complete* pasamos al estado **__READ_RESP__**.

Close en origin

- **READ_RESP**: Desregistramos al origin del selector y vamos al estado **__NO_ORIGIN__**.

Escribir en transformacion

- **READ_RESP**: Si hay caracteres en *pre_trans_buffer*, los enviamos al programa de transformación.

Leer en transformacion

- **READ_RESP**: Si hay espacio en *post_resp_buffer*, leemos del programa de transformación al buffer *post_resp_buffer*.

2.4 Modulo request_parser

Este módulo se encarga de leer lo que se encuentra en un buffer de entrada (a la salida del cliente), parsearlo para extraer información relevante del request como el host y el método http y dejarlo en un buffer de salida para que se lea del origin una vez encontrado el host. Para esto último se llama a un callback dentro del parser.

Adicionalmente, el parser internamente maneja estados de manera de reanudar la operación si en algún momento se encuentra que el buffer de entrada está vacío y se guarda la información necesaria para reanudar la operación. Por ejemplo, si estoy verificando el método y me llega GE nada más al buffer de entrada luego cuando me llega una T se reanuda la operación y el parser sabe identificar que el método es GET.

Como función extra, el parser también agrega un buffer artificial (X-LOCALHOST) a la lista de headers del request para verificar que no tenga un loop. En caso de tener uno, en el siguiente llamado al request parser cuando encuentro dicho header seteo una variable que me indica que tengo un loop.

<u>Estados</u>

Los estados que maneja el módulo en su máquina de estados son:

- **SPACE_TRANSITION**: Para cuando tengo una transición con espacios entre dos estados.
- **METHOD**: Para verificar el método http.
- **URI**: Para verificar el uri. Para este caso lo que quiero ver es si tengo un uri relativo o uno absoluto. Luego pasó al estado correspondiente.
- **RELATIVE_URI**: Cuando estoy en el estado de uri relativo lo único que me interesado es recorrer el buffer hasta llegar a la transición a la versión http.
- **URI_HOST**: Cuando estoy en el estado de uri absoluto me interesa extraer el host y potencialmente el puerto. Si encuentro el host llamó al callback correspondiente desde la máquina de estados.
- **VERSION**: Para verificar la versión http.
- **START_LINE_END**: Para verificar el \r\n entre el start line y los headers.
- **LOCALHOST_HEADER_CHECK**: Para verificar si tengo el header X-LOCALHOST a la entrada.
- **HEADERS**: Cuando estoy verificando headers.
- **HOST**: Cuando estoy verificando el contenido del header Host. Si encuentro el host llamó al callback correspondiente desde la máquina de estados.
- **LENGTH**: Cuando estoy verificando el contenido del header Content-length.
- **CHUNKED**: Al verificar el contenido del header Transfer-encoding quiero ver si aplica la directiva chunked.
- **BODY**: Cuando estoy parseando el contenido del body habiendo pasado el doble CRLF final de los headers.
- **FINISHED**: Cuando o bien terminé de leer el body o llegué al final de headers sin tener un content-length o un transfer-encoding: chunked. También pongo en estado FINISHED cuando encuentro un LOOP.

2.5 Modulo response_parser

Este módulo se encarga de leer lo que se encuentra en un buffer de entrada (a la salida del origin), parsearlo para extraer información relevante del response como el body y el content-type y dejarlo o bién en un buffer de salida (a la entrada del client) o en un buffer de transformación (a la entrada del programa de transformación) si es que el media type machea con alguno de mi lista.

Adicionalmente, como para el request el parser internamente maneja estados de manera de reanudar la operación si en algún momento se encuentra que el buffer de entrada está vacío y se guarda la información necesaria para reanudar la operación.

También, en el caso en que detectó que voy a tener transformaciones agrego artificialmente el header "Transfer-encoding: chunked" si que es ya no está presente de manera de permitir que el programa de transformación pueda devolver el body transformado chunkeado.

Estados

Los estados que maneja el módulo en su máquina de estados son:

- **RES_SPACE_TRANSITION**: Para cuando tengo una transición con espacios entre dos estados.
- **RES_VERSION**: Para verificar la versión http.
- **STATUS**: Para verificar el estado de la respuesta.
- **RES_HEADERS**: Cuando estoy verificando headers.
- **LENGTH_CHECK**: Cuando estoy verificando el contenido del header Content-length.
- **ENCODING_CHECK**: Cuando estoy verificando el contenido del header Content-length.
- **CHUNKED_CHECK**: Al verificar el contenido del header Transfer-encoding quiero ver si aplica la directiva chunked.
- **CONNECTION_CHECK**: Cuando encuentro el header Connection quiero cambiar su directiva a close porque no manejo conexiones persistentes.
- **BODY_NORMAL**: Cuando estoy parseando el contenido del body sin transformaciones. En este caso lo que hago es verificar si termino de leer el body y voy pasando todo al buffer de salida.
- **BODY_TRANSFORMATION**: Cuando estoy parseando el contenido del body con transformaciones. En este caso lo que hago es verificar si termino de leer el body y voy pasando todo al buffer de transformación. Si lo que tengo en el response está en formato chunked voy extrayendo los chunks y se los voy pasando al buffer de transformación.
- **RES_FINISHED**: Cuando o bien terminé de leer el body o llegué al final de headers sin tener un content-length o un transfer-encoding: chunked.

3. Problemas encontrados durante el diseño y la implementación

A continuación se presentan algunos de los errores encontrados durante el diseño y la implementación del trabajo.

Funciones deprecated con alternativa no implementada en linux

Un problema hallado fue al implementar tanto el cliente como el servidor administrador con el uso de sctp. Las funciones otorgadas por este para el envío y recepción del mensaje se encuentran obsoletas (sctp_recvmsg, sctp_sendmsg) según el RFC6458:

> *"(...)**9.7. sctp_sendmsg() - DEPRECATED***
> *This function is deprecated; sctp_sendv() (see Section 9.12) should be used instead.(...)"*
>
> *"(...)**9.8. sctp_recvmsg() - DEPRECATED***
> *This function is deprecated; sctp_recvv() (see Section 9.13) should be used instead.(...)"*

Creímos conveniente utilizar estas nuevas funciones indicadas, pero no pudimos ya que no tienen una implementación en la librería de sctp (lksctp) Linux.

Uso tardío de ASAN durante la implementación

Nos ocurrió que muchos errores fueron detectados únicamente al compilar con "-fsinitizer=address". Previo a hacerlo, creímos que muchas cosas funcionaban correctamente, pero luego fue mucho más costoso que lo que hubiera sido si desde un principio hubiéramos compilado con esta herramienta.

Intereses mal seteados

Tuvimos problemas inicialmente al setear mal los intereses de cada file descriptor en el selector, esto ocasionaba que se llame al handler de manera repetida cuando no había acciones para realizar lo cual generaba un gran uso de la CPU.

4. Limitaciones de la aplicación

- Al enviar una response con “Transfer-Encoding = Chunked”, a pesar de que no se tiene en cuenta el header “Content-Length’, este sigue estando en lugar de ser eliminado.
- Una de las grandes limitaciones de la aplicación y por la cual no se cumple un objetivo principal es que las transformaciones no se realizan al detectar un media type. Si bien están implementados los mecanismos para realizarlas, actualmente sólo se puede hacer cambiando un flag en el parser de la response lo cual implica una recopilación del tp. Otra opción es dejar este flag seteado y a modo de “apagar” las transformaciones, setear “cat” como programa de transformación. Esta solución no es la adecuada pero se propone como una forma de probar que el mecanismo funciona y se puede cambiar el programa en tiempo de ejecución. Por otro lado, se encuentra en la rama parsers-adapted una versión donde funcionan las transformaciones pero por problemas de tiempo no llegamos a mergear estos cambios a master.
- El admin-client tiene ciertos bugs: Al pedir la lista de configuraciones o de métricas, muchas veces muestra algunos valores repetidos. Muy de vez en cuando la aplicación falla y no devuelve el valor esperado, comenzando a responder a la petición anterior en lugar de a la actual.
- Al parsear una request en búsqueda del header host (cuando aún no se tiene el host), como aún no hay origin que consuma del buffer, el host se debe encontrar dentro de los BUFFER_SIZE (obtenido desde el módulo de configuración) caracteres de la request.
- Hay configuraciones (como los puertos) para las cuales no se debería permitir modificar sus valores en tiempo de ejecución. Actualmente permite modificar el valor, pero este no tiene efecto alguno. Esto podría chequearse al momento de setear, enviar un error de un nuevo tipo “CONFIG_SET_NOT_ALLOWED” o no disponibilizar este tipo de configuraciones al cliente.
- En cuanto al estilo de programación, no hubo un acuerdo grupal en cuanto a usar camel case o snake case para los identificadores, es por eso que en su mayoría, el código utiliza snake case pero los parsers utilizan camel case.

5. Posibles extensiones

Las posibles mejoras o extensiones que creemos que se puede hacer al trabajo son:

- Incluir la posibilidad de enviar las credenciales para autorizar el acceso a métricas y configuraciones
- Realizar una abstracción de los distintos componentes del flujo de datos de un cliente. En esta abstracción e interfaz se podrían indicar cuáles componentes (parsers, cliente, origin, transformacion, etc.) producen y cuales consumen. De esta forma, se tiene una organización más uniforme sobre los datos disminuyendo la posibilidad de bugs. Esto además trae como ventaja que se hace más sencillo interconectar los distintos componentes entre sí.
- Finalizar las características que se piden en el enunciado y que no fueron implementadas
- Estudiar el uso de sendfile para transmisiones sin parseo ni transformaciones
- Solventar la limitación de filedescriptors.
- Revisar el parseo de request y responses
- Incluir el módulo HTTPResponses que genera los mensajes de error a demanda.

6. Conclusiones

El desarrollo del trabajo nos permitió aprender de manera práctica las implicaciones de una aplicación cliente-servidor. Consideramos que fue importante tener un planeamiento previo para realizar la implementación de la manera más limpia posible.

La realización del trabajo implicó un gran cambio y aprendizaje en la forma de diseñar la arquitectura de las aplicaciones. Pudimos ver cómo es posible obtener concurrencia utilizando solo un thread para las tareas principales, y la importancia de la organización en el manejo de eventos e intereses para garantizar un buen rendimiento.

En cuanto al parseo, tuvimos que realizarlo a demanda y con la posibilidad de que sea byte a byte, esto fue una de las dificultades más grandes del trabajo debido a la gran cantidad de chequeos realizados en estos. Creemos que tal vez hubiera sido mejor realizar un parser con las verificaciones imprescindibles para nuestro uso y dejar el resto de los chequeos al servidor origin.

Por otro lado, aprendimos otro aspecto en cuanto a diseño al momento de pensar el protocolo de comunicación entre el servidor proxy y el cliente de administración. En esta etapa del trabajo valoramos mucho haber tenido contacto previamente con los RFC ya que pudimos sacar ideas de otros protocolos.

Algo que se podría cambiar en cuanto a la actitud durante el desarrollo del trabajo es que dar mucho detalle o considerar muchas alternativas a la hora de tomar una decisión nos llevó a mal gastar el tiempo y por consecuencia no completar todos los objetivos del trabajo.

7. Instrucciones de instalación

Las dos aplicaciones que conforman el TPE, están organizadas como proyectos separados. Se utiliza la herramienta de manejo de dependencias y configuración CMake. A continuación se muestra cómo compilar cada una.

7.1 Compilacion de proxy_http

Para compilar proxy_http se debe ejecutar ./compile.sh dentro del directorio proxy_http. Este script genera el directorio build y posteriormente, dentro de él configura y compila el proyecto. Si la compilación sucedió con éxito, se escriben los ejecutables dentro de proxy_http/build/target, tanto la aplicación como los tests.

7.2 Compilación de admin_client

El cliente de administración se compila de forma similar al proxy. Dentro del directorio admin_client se encuentra un script compile.sh que realiza los mismos pasos que el de http_proxy, dejando la aplicación en el directorio admin_client/build/target.

Adicionalmente, creímos conveniente agregar un script para generar nuevos módulos de forma automática. Este script es new_module.sh y se encarga de crear los directorios y archivos necesarios asi como tambien se agrega como dependencia de la aplicación.

8. Configuración y monitoreo

Para crear las configuraciones deseadas se debe modificar en el main.c del “proxy_http” la seccion *config init.* En esta se llama a la función “config_create”, la cual recibe dos char *: *un nombre y un valor.* Si se desea tener otras configuraciones, o modificar el valor de inicio de alguna de las establecidas, se deben cambiar estos llamados.

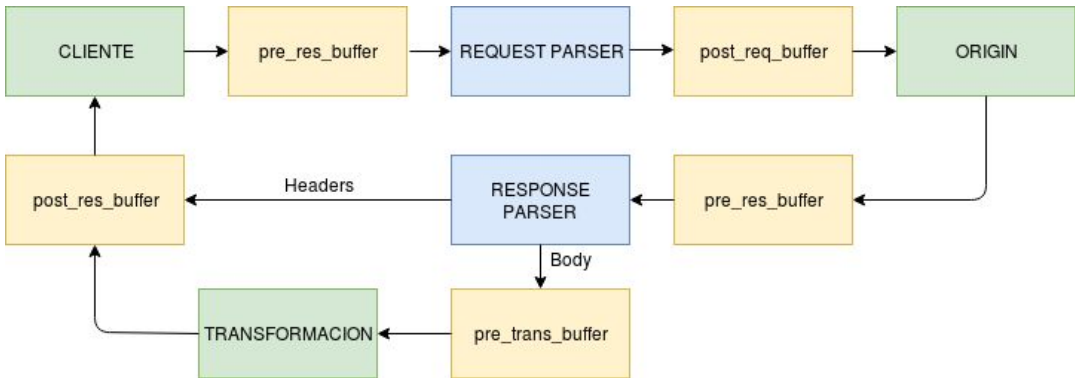
Para consultar métricas y configuraciones, y setear configuraciones, se debe ejecutar el programa “admin_client”.
Cuando este empieza a correr, se presentan diferentes funcionalidades:
1) Listar metricas
2) Listar configuraciones
3) Obtener metrica (indicar numero de metrica)
4) Obtener configuración (indicar numero de configuración)
5) Setear configuración (indicar numero de configuración y valor deseado
6) Cerrar
Tanto “Listar métricas” como “Listar configuraciones”, muestran una lista en la cual cada nombre y valor va acompañado de un número. Este número es el que se utiliza luego en las opciones 3, 4, 5.
Las métricas se inicializan al inicio del programa y luego se actualizan constantemente, como se indica en la explicación del modulo métricas.
En el caso de las configuraciones se inicializan al principio del programa. Luego mediante la ejecución de admin_client, sus valores pueden ser modificados.

9. Arquitectura de la aplicación

La aplicación proxy_http en cuanto a organización de código se encuentra divida en módulos, cada uno cumple una función específica. Los módulos utilizados son:

- Archivo main
- Client: mantiene registro del estado actual de un cliente, e información relevante para este, como por ejemplo, buffers, parsers, file descriptors tanto del servidor origin como de las transformaciones.
- RequestParser: parser a demanda de requests HTTP. Permite extraer información como por ejemplo, el host del origin server.
- ResponseParser: parser a demanda de responses HTTP. Permite extraer información como el Content-Type para saber si se debe aplicar una transformación.
- Buffer: implementación de buffer provista por la cátedra.
- Selector: implementación de un selector (abstracción de select) para el manejo no bloqueante de file descriptors.]
- Config: centraliza todas las configuraciones que puedan ser cambiadas en tiempo de ejecución. Permite almacenar configuraciones las cuales se identifican por un string.
- Metrics: Permite calcular métricas basándose en distintos eventos llamados durante la atención de un cliente.
- Logger: Permite mantener un registro de los distintos eventos realizados en el proxy.
- HTTPResponses: permite escribir responses http en un buffer a demanda. Si bien esta realizada la implementación, por cuestiones de tiempo, no se llegó a utilizarla.
-

En cuanto al manejo de datos, se utiliza principalmente una entidad cliente la cual almacena el estado de cada cliente conectado. Cada cliente, al ser aceptado es agregado junto con el file descriptor de su socket, al selector, el cual se encarga de manejar los intereses y eventos de escritura y lectura de cada uno.
Cada cliente tiene una serie de buffers los cuales sirven de nexo entre los distintos productores y consumidores presentes en la atención. Estos componentes son, el cliente, opcionalmente el programa de transformación y finalmente el host origin. En esta ultima categoria tambien se incluiría el HTTPResponses que no llegamos a incluir. Luego se encuentran los parsers los cuales leen de un buffer y escriben a otro. Se puede ver un grafico de lo explicado a continuación.



10. Pruebas de tiempo de transferencia e integridad de datos

Transferencias de origin a client
A continuación se muestran las pruebas realizadas para ver que el proxy no altere los datos y tener noción de los tiempos con distintas optimizaciones y compiladores. Más abajo se encuentra una tabla con los tiempos de transferencias.

Sin proxy:
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.09s user 0.26s system 19% cpu **1.780 total**
sha256sum 1.71s user 0.04s system 98% cpu 1.779 total

10.1 Compilado con GCC

Con proxy:
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.16s user 0.36s system 4% cpu **11.380 total**
sha256sum 2.26s user 0.08s system 20% cpu 11.380 total

Con proxy **compilado con O1:**
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.10s user 0.23s system 8% cpu **3.777 total**
sha256sum 2.34s user 0.06s system 63% cpu 3.776 total

Con proxy **compilado con O2**:
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.11s user 0.24s system 9% cpu **3.625 total**
sha256sum 2.27s user 0.06s system 64% cpu 3.624 total

Con proxy **compilado con O3**:
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.11s user 0.24s system 10% cpu **3.399 total**
sha256sum 2.22s user 0.08s system 67% cpu 3.398 total

10.2 Compilado con Clang

Con proxy:
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.16s user 0.34s system 5% cpu **9.798 total**
sha256sum 2.27s user 0.09s system 24% cpu 9.797 total

Con proxy **compilado con O1:**
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.11s user 0.28s system 6% cpu **5.600 total**
sha256sum 2.30s user 0.08s system 42% cpu 5.599 total

Con proxy **compilado con O2**:
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.10s user 0.24s system 10% cpu **3.209 total**
sha256sum 2.28s user 0.09s system 73% cpu 3.208 total

Con proxy **compilado con O3**:
time curl -s http://localhost/big.tar.gz | sha256sum
8a2f58ca3f7ebcbe0e4c256d7478acf34cbecc65197612b10df5af05f26a1e95 -
curl -s http://localhost/big.tar.gz 0.12s user 0.22s system 10% cpu **3.227 total**
sha256sum 2.28s user 0.07s system 72% cpu 3.226 total

Tabla de resultados:

Proxy	GCC [s]	CLANG [s]
NO	1.780	
SI	11.380	9.798
SI (-O1)	3.777	5.600
SI (-O2)	3.625	3.209
SI (-03)	3.399	3.227

Como se puede observar, se obtiene un incremento grande en las velocidades de transferencia al usar las optimizaciones del compilador, llegando a que el tiempo de transferencia pasando por el proxy, sea menor al doble que sin usar el proxy.
Aclaraciones: los tiempos expuestos son resultado de **una** ejecución, pero fueron realizadas varias pruebas y se pudo observar que las variaciones están en el orden de 10ms.