

# Informe final

# Proxy HTTP

---

Biagini, Martín

Cortés Rodríguez, Kevin

Filipic, Joaquín

Nielavitzky, Jonathan

ITBA - Protocolos de Comunicación - 1º Cuatrimestre 2018

Martes, 12 de Junio

# Índice de contenidos

<b>Índice de contenidos</b>	<b>1</b>
<b>Introducción</b>	<b>2</b>
<b>Proxy</b>	<b>3</b>
Descripción funcional básica	3
Argumentos para la ejecución	4
<b>Principales funciones</b>	<b>6</b>
main	6
cbuffer	7
child_process	8
clients_manager	9
clients	10
config_client	11
config_socket	13
master_socket	14
to_origin_server	15
utilities	16
commons	17
parser_definitions	18
parser_utils	19
parser	20
store_manager	22
<b>Diagrama de secuencia</b>	<b>23</b>
<b>Monitoreo/Configuración</b>	<b>24</b>
Protocolo de Transporte: SCTP	24
Protocolo de Aplicación	25
Estado NOT_AUTH	26
Estado AUTH	27
Estado QUIT	31
Ejemplo de uso	31
<b>Problemas encontrados en general</b>	<b>33</b>

## Introducción

El siguiente informe tiene por objetivo demostrarle a la cátedra la capacidad del equipo para programar un proxy HTTP versión 1.1 (RFC 7230) y la habilidad para la programación de aplicaciones cliente/servidor con sockets, la comprensión de estándares de la industria, y la capacidad de diseñar un protocolo de aplicación basado sobre el protocolo de transporte SCTP.

Por otro lado, se detallarán algunas cuestiones de implementación, estudio y análisis de los métodos y clases más importantes, análisis de la máquina de estados para el parseo de la información (request y response del origin server) y del protocolo SCTP.

**Aclaración:** Todas las pruebas se realizaron con el servidor brindado por la cátedra. Para hacerlas, usted puede optar por levantarlo localmente o conectarse al servidor <http://54.233.125.152:9090/> hospedado en AWS por el equipo.

A su vez, también puede hacer uso del proxy implementado accediendo a la ip [54.233.99.97](http://54.233.99.97) a los puertos por default establecidos (8080 para el proxy, 9090 para la configuración del cliente).

Para las pruebas de archivos grandes, puede entrar al contexto /web\_template.tar

### Código Fuente:

- <https://bitbucket.org/itba/pc-2018-02/src/master/>
- <http://54.233.125.152:9090/codigo-fuente.zip>

### Informe:

- <http://54.233.125.152:9090/informe.pdf>

### Diagrama de secuencia:

- <http://54.233.125.152:9090/diagrama-1.html>

## Proxy

### Descripción funcional básica

El proxy HTTP funciona en base a `pselect()` (tiene un máximo teórico de conexiones soportadas, 1024), que se encuentra dentro de un ciclo `while` constante, manejando las conexiones concurrentemente.

Inicialmente, el programa setea todas las configuraciones necesarias y recibe los argumentos debidos, para luego entrar en la iteración del mencionado ciclo. Dentro del mismo, lo primero que se hace es resetear los sets de file descriptors (correspondientes a los sockets que se utilizan para conectarse con clientes y servidores) tanto de lectura como de escritura suscriptos al `pselect`, y luego se añaden sólo aquellos que deban tenerse en cuenta (actualizándose al manejarse las operaciones de lectura y escritura de cada cliente y servidor).

A continuación, el `pselect` entra en estado de bloqueo hasta desbloquearse frente a algún cambio en algún file descriptor. El desbloqueo puede deberse a un nuevo pedido de conexión o a un requerimiento de lectura o escritura en un socket. Vale mencionar que se hace uso de 2 sockets pasivos (uno para las conexiones de clientes HTTP, que derivan las conexiones entrantes a sockets activos, y otro para el monitoreo/configuración).

En cuanto se recibe una conexión de un cliente HTTP, se analizan el host y el puerto de la petición y se abre un hilo para poder realizar la resolución de nombre de dicha dirección. Cuando el hilo finaliza, envía una señal a su padre para indicárselo y poder establecer la conexión con el origin server.

Si el desbloqueo del `pselect` se debe a una operación de lectura o escritura, se deriva su manejo a distintas funciones puntuales que resuelven cada pedido. En cuanto a enviar la información de un lado a otro, se utiliza la estructura de buffers provista por la cátedra.

Y con respecto a la transformación externa de las respuestas, lo que se hace es abrir un proceso hijo (con el cual se comunica mediante pipes, uno para escribir y otro para leer) y derivarle la data. Este proceso se encarga de llamar al intérprete `bash` e indicarle que corra el comando preestablecido para aplicar las transformaciones. Cabe destacar que esto se realiza si las transformaciones están activadas y si el Media Type de la respuesta HTTP a transformar está incluida en la lista configurable de Media Types transformables.

## Argumentos para la ejecución

Para correr compilar el proxy, deberá correr el makefile presente en el repositorio.

<code>make</code>	Salida: un archivo ejecutable <code>./proxy</code>
<code>make clean</code>	Borra los ejecutables creados anteriormente.

Ejecutando el archivo proxy, el mismo correrá en el puerto 8080 para el proxy en sí; y el 9090 para el cliente de administración SCTP escuchando en la interfaz de loopback. Éstos son los valores por default.

En caso de querer cambiarlos, se puede ingresar cualquiera de las siguientes opciones:

<code>-h</code>	Imprime la ayuda y termina.
<code>-l dirección-http</code>	Establece la dirección donde servirá el proxy HTTP. Por defecto escucha en todas las interfaces.
<code>-L dirección-de-management</code>	Establece la dirección donde servirá el servicio de management. Por defecto escucha únicamente en loopback.
<code>-M media-types-transformables</code>	Lista de media types transformables. La sintaxis de la lista sigue las reglas del header Content-Type de HTTP. Por defecto la lista se encuentra vacía. Por ejemplo el valor <code>text/plain,image/*</code> transformará todas las respuestas declaradas como <code>text/plain</code> o de tipo imagen como <code>image/png</code> .
<code>-o puerto-de-management</code>	Puerto SCTP donde se encuentra el servidor de management. Por defecto el valor es 9090.
<code>-p puerto-local</code>	Puerto TCP donde escuchara por conexiones entrantes HTTP. Por defecto el valor es 8080.
<code>-t cmd</code>	Comando utilizado para las transformaciones externas. Compatible con <code>system(3)</code> . La sección <code>FILTROS</code> describe cómo es la interacción entre <code>httpd(8)</code> y el comando <code>filtro</code> . Por defecto no se aplica ninguna transformación.



-v	Imprime información sobre la versión version y termina.
----	---

## Principales funciones

### main

#### Objetivo de main.c:

Contiene la función principal. Inicialmente setea los valores en una estructura (definida en settings.c) con los valores por default establecidos por la cátedra en el manual httpd.8. En caso de querer cambiar los valores, el responsable de leerlos será la función handle\_main\_options.

Crea la estructura de una lista de clientes. Dicha lista es la que va a contener a los clientes que se conectan al proxy. Los clientes pueden ser clientes http o clientes de administrador.

Inicializa el master\_socket y el config\_socket. Uno responde a las peticiones de clientes http y el otro al cliente de configuración y administración SCTP.

#### Funciones que abarca:

- void resolve(clients\_list \*cl, fd\_set readfds, fd\_set writefds, settings \*st)

Resuelve las peticiones de los clientes dentro de la lista de clientes.

- void handle\_signal\_action(int sig\_number)

Si recibe alguna señal, corta la ejecución y cierra correctamente la conexión con todos los clientes en la lista.

- void shutdown\_properly(void)

Cierra todos los filedescriptors de los clientes agregados a la lista.

- int setup\_signals(void)

## cbuffer

### Objetivo de cbuffer.c:

Este buffer fue provisto por la cátedra. El objetivo es tener un buffer circular, que usa internamente punteros de lectura y escritura. Cada vez que se quiere hacer una escritura en el buffer, se escribe y se hace un avance del puntero de escritura. Si se quiere leer, lo mismo ocurre pero con el puntero de lectura. Siempre se tiene en cuenta la condición que los punteros estén iniciados en la posición 0 y que la posición del puntero de lectura sea menor o igual a la posición del puntero de escritura.

### Funciones que abarca:

```
struct buffer {  
    uint8_t *data;  
    uint8_t *limit; → límite superior del buffer. inmutable.  
    uint8_t *read; → puntero de lectura.  
    uint8_t *write; → puntero de escritura.  
};
```

- **buffer\_read**(buffer \*b);

Obtiene un byte del buffer, dependiendo de donde esté el puntero de lectura.

- **buffer\_write**(buffer \*b, uint8\_t c);

Escribe un byte en el buffer, dependiendo de la posición del puntero de escritura.

- **doread**(int \*fd, struct buffer \*buff);

Lee del buffer hasta el puntero de escritura (ya que el puntero de lectura no puede sobrepasar al de escritura)

- **dowrite**(int \*fd, struct buffer \*buff);

Escribe en el fd lo que haya en el buffer.

- **buffer\_write\_ptr**(buffer \*b, size\_t \*nbyte);
- **buffer\_write\_adv**(buffer \*b, const ssize\_t bytes);
- **buffer\_read\_ptr**(buffer \*b, size\_t \*nbyte);
- **buffer\_read\_adv**(buffer \*b, const ssize\_t bytes);

Son funciones para avanzar los punteros, tanto de lectura como de escritura.



## child\_process

### Objetivo de child\_process.c:

Crea el proceso hijo para que pueda aplicar la función transformadora. Dicha función se ejecuta en `init_child`, y es llamada cuando las transformaciones estén activadas y el media-type a que devuelve la response del origin server, corresponde con uno de los de la lista.

La función transformadora se ejecuta en el proceso hijo, y la función se pasa por parámetro en el cliente de configuración. Internamente se hace una llamada a

```
execl("/bin/sh", "sh", "-c", function, NULL)
```

donde “function” es la función que transformará.

### Funciones que abarca:

- `int init_child(child_process *cp);`

Inicia el proceso hijo y setea los pipes correspondientes.

- `void set_fd_for_child(child_process *cp, int *max_fd, fd_set *readfds, fd_set *writefds, struct buffer *write_to, struct buffer *read_from);`

Setea los valores de los file descriptors, dependiendo si el proceso hijo necesita leer o escribir. También recibe por parámetro los buffers a los que tiene que leer y a donde tiene que escribir. La idea es que sea posible pasar el buffer de request del cliente o del server y usarlo como un buffer de lectura, para que cuando el proceso hijo tenga que transformar el contenido del buffer, lo escriba en un tercer buffer.

- `void resolve_child(child_process *cp, fd_set *readfds, fd_set *writefds, struct buffer *write_to, struct buffer *read_from);`

Cuando el selector se desbloquea, resuelve lo que tiene que hacer del cliente.

- `void free_child(child_process *cp);`

## clients\_manager

### Objetivo de clients\_manager.c:

En simples palabras, es quien se encarga de tomar las decisiones de lectura y escritura para cada cliente para el manejo del selector. Además maneja otras cuestiones relacionadas específicamente del cliente; como por ejemplo, la resolución de la IP y puerto del servidor al que se quiere conectar (origin server).

### Funciones que abarca:

- `int set_fd_for_client(client *c, clients_list *cl, int *max_fd, fd_set *readfds, fd_set *writefds, settings *st);`

Es quien maneja los FD\_SET para que luego, en la próxima iteración, sea “handleado” por el selector. Se enlista para lectura o escritura siempre y cuando pueda leer del buffer o escribir al buffer (buffer del cliente o del servidor). También verifica el estado de la conexión con el origin server. Hay casos de que cuando ocurre algún error, se envía un mensaje 503 (Service Unavailable) y se desconecta al usuario.

- `void accept_new_client(clients_list *cl, int master_socket, struct sockaddr_in address, int addrlen, settings *st);`

Acepta un nuevo cliente y lo agrega a la lista.

- `void* placeholder (void *args);`

Función que resuelve al IP del hostname. Esta función corre en un thread aparte, ya que la función getaddrinfo es bloqueante. Para “avisar” de que la resolución del nombre ya finalizó, se envía una señal al selector.

- `int client_resolved_server(client *c);`
- `void resolve_client(client *c, clients_list *cl, fd_set *readfds, fd_set *writefds, settings *st);`

Hace la función FD\_ISSET. Lee de un buffer y escribe en otro.

## clients

### Objetivo de clients.c:

Contiene la información relevante para manejar al cliente por el selector; y otros datos de interés. El `clients_list` es una lista doblemente encadenada de éste tipo de estructura.

Dentro de sus características principales, contiene los buffers en donde: recibirá la información, enviará la información, buffer que deposita la información transformada, utilizada por la función transformadora. También contiene una estructura particular para las requests y response.

### Funciones que abarca:

- `clients_list *init_client_list(clients_list *cl);`

Inicializa la estructura de lista del cliente.

- `client *new_client(clients_list *cl, int fd);`

Crea un nuevo cliente, pasandole por parámetro su file descriptor correspondiente; y lo agrega a lo último de la lista.

- `int remove_client(clients_list *cl, int fd);`

Remueve al cliente de la lista y libera los recursos. Esta función es usada cada vez que se desconecta el cliente.

- `void client_is_transforming(client *c);`

Función utilizada cuando la opción de transformación está activada. Adentro de ella, se llama a la función `init_child`; mencionada anteriormente.

## config\_client

### Objetivo de config\_client.c:


Este archivo es una implementación del protocolo de configuración y que permite monitorear en tiempo real el proxy. Al compilarse se genera el ejecutable **config\_client**, que al correrlo recibe por argumentos las operaciones que se desean realizar y se comunica con el proxy via SCTP para aplicar configuraciones y obtener métricas. Inicialmente, se analizan las opciones ingresadas como parámetros y se encarga de derivarlas a las funciones que construyen los comandos según el protocolo.

A continuación, la lista de posibles argumentos que recibe el programa:

-a	Activar la transformación externa de Media Types.
-d	Desactivar la transformación externa de Media Types.
-M media-types-transformables	Lista de media types transformables. Por defecto la lista se encuentra vacía. Por ejemplo el valor text/plain,image/* transformará todas las respuestas declaradas como text/plain o de tipo imagen como ser image/png.
-r	Resetear la lista de Media Types transformables.
-t cmd	Seteo del comando utilizado para las transformaciones externas. Compatible con system(3).
-1	Obtener cantidad de conexiones concurrentes.
-2	Obtener cantidad de accesos históricos.
-3	Obtener cantidad de bytes transferidos de parte del cliente.
-4	Obtener cantidad de bytes transferidos de parte del servidor.

### Funciones que abarca:

- static void **createMessageToSend**(const char\* pref, const char\* msg);



Para los comandos con argumentos, se construye el mismo concatenando el mismo luego del caracter inicial del comando y el espacio.

- static void **sendAndReceive**(const char\* msg);

Envía el comando construido al proxy y recibe su respuesta.

- static void **checkAuthState**(const char \* recv\_buffer);

Para evitar enviar comandos sin estar autenticado, se analiza la respuesta del servidor al comando de envío de contraseña. En caso de fallar, no se enviarán los comandos subsiguientes.

## config\_socket

### Objetivo de config\_socket.c:

Contiene funciones para seteos e inicializaciones del socket pasivo de configuración, así como llamadas a las funciones para parsear los pedidos.

### Funciones que abarca:

- void **init\_config\_socket**(int \*conf\_sock, settings \*st);

Setea las configuraciones básicas del conf\_sock (socket pasivo que acepta las conexiones entrantes de clientes de configuración). El valor del puerto en donde escuchar es definido en esta función, obtenido de la estructura settings.

- void **init\_config\_options**(int conf\_sock);

Configuración de los servicios utilizados provistos por el protocolo SCTP (multihoming y notificaciones).

- void **set\_fd\_for\_config\_socket**(const int \*conf\_sock, int \*max\_fd, fd\_set \*readfds);

Inscribe como lectura al master socket de configuración para el selector.

- void **resolve\_config\_client**(int conf\_sock, fd\_set \*readfds, settings \*st);

Cada vez que un cliente de configuración hace un pedido, se lo deriva a las funciones de parseo para saber qué responder a cada comando.

### Decisiones de implementación tomadas:

Actualmente, el servidor mantiene un socket pasivo para escuchar a los clientes, pero no los deriva a otro socket cuando alguien se conecta, si no que está configurado para responderle sabiendo su struct sockaddr (que la obtiene al recibir el mensaje con stp\_recvmmsg()). La conexión se cierra cuando el cliente se desconecta, ya que el servidor va a quedar siempre escuchando.

Como extensión, puede añadirse la implementación de que el servidor cierre la conexión cuando el cliente le envíe el comando de cierre, para acelerar el proceso de desconexión.

## master\_socket

### Objetivo de master\_socket.c:

#### Funciones que abarca:

- void **init\_master\_socket**(int \*master\_socket, struct sockaddr\_in \*address, int \*addrlen, settings \*st);

Setea las configuraciones básicas del master\_socket (es al que los clientes se conectarán). Los valores del puerto y de las interfaces son especificadas en esta función. Dichos valores son obtenidos de la estructura settings, iniciada al comienzo de la ejecución del proxy.

- void **set\_fd\_for\_master\_socket**(const int \*master\_socket, int \*max\_fd, fd\_set \*readfds);

Inscribe como lectura al master socket para el selector.

- void **resolve\_master\_client**(int master\_socket, fd\_set \*readfds, clients\_list \*cl, struct sockaddr\_in address, int addrlen, settings \*st);

Chequea si hay nuevos clientes.



## to\_origin\_server

### Objetivo de origin\_server.c:

Es quién se encarga de establecer la conexión con el origin server.

### Funciones que abarca:

- int **connect\_server**(client \*c);

Recibe por parámetro al cliente, ya que en su estructura dice la IP y puerto al que se quiere conectar del origin server. Para ello, se utiliza la función “connect”, y hay que setear, primero que todo, los flags para que sea no bloqueante.



## utilities

### Objetivo de utilities.c:

Contiene información extra para el correcto desempeño del parser, cumpliendo con las especificaciones brindadas por la cátedra.

### Funciones que abarca:

- `char* response_503(void);`

Devuelve un string con formato HTTP simulando ser una respuesta 503 de Service Unavailable. Dicha respuesta se dará al usuario por parte del proxy cuando ocurra cualquiera de los tres escenarios:

- a. Hostname no existe
- b. Intento de conexión fallido
- c. Timeout

- `int isMediaTypeInList(char *responseMType, struct settings *st);`

Esta función es meramente para saber si tenemos que aplicar la función transformadora cuando recibimos la respuesta del origin server, después de que sus headers hayan sido parseados por el `parser_response`.

- `int compareMediaType(char *responseMType, char *setMType);`



## commons

### Objetivo de commons.c:

El objetivo es implementar las funciones necesarias para identificar correctamente los bytes que pasan por el proxy y requieren ser analizados. Las funciones corresponden con los diferentes tipos definidos en los RFC 2234, 3986 y 7230 que son utilizados por los mismos para las definiciones de los elementos HTTP.

### Decisiones de implementación tomadas:

Se tomó la decisión de implementar funciones que ya encuentran implementadas y son de uso común, como las funciones *isalpha*, *isdigit*, *isxdigit* e *isspace*. De esta manera, la persona tiene a su alcance la implementación de todas y cada una de las funciones necesarias para el correcto análisis de los elementos HTTP.

## parser\_definitions

### Objetivo de parser\_definitions.h:

Definir las estructuras utilizadas para almacenar la información parseada por el proxy tanto para los mensajes de request como para los de response. Adicionalmente, se definen los enums necesarios para identificar los estados de las distintas máquinas de estados utilizadas por las funciones encargadas de parsear los mensajes.

### Estructuras más importantes:

*header\_line*: contiene dos strings (char \*) correspondientes al *header\_name* y *header\_value*.

- *status\_line*: incluye strings de los distintos elementos que se hallan en la primer línea de los mensajes HTTP, según el RFC 7230.
- *http\_message*: incluye una estructura *status\_line*, un array de estructuras *header\_line* y la cantidad de elementos que contiene dicho array.
- *request\_parser\_info*: contiene dos punteros a estructuras de tipo *buffer*. El primero, *aux\_buffer*, se utiliza para guardar los elementos que están siendo parseados. Esto es necesario por si se termina de leer el contenido de un paquete de bytes a la mitad de un *header\_name*, por ejemplo. El segundo, *headers\_buffer*, se utiliza para guardar temporalmente el contenido de la *request*, hasta que sea posible resolver la dirección y el puerto del *origin server*. Por otro lado, incluye un puntero a la estructura *http\_message*; un enum de tipo *state* que almacena el estado actual de la máquina de estados del parser; un booleano, *raining*, que indica si está activado el modo “pasamanos”. Esto significa que el proxy deja de parsear los bytes recibidos, y los redirige directo al *origin server*. También se incluye un booleano, *ready\_to\_connect*, que se setea en true cuando se obtuvieron la dirección y el puerto del *origin server*.
- *response\_parser\_info*: contiene elementos similares a la estructura anterior, agregando algunos campos necesarios para el parseo de una respuesta con *transfer-encoding: chunked*.

## parser\_utils

### Objetivo de parser.c:

Proveer tanto al proxy como al parser de funciones útiles que permitan interactuar con las estructuras definidas en el archivo *parser\_definitions.h*.

### Funciones más relevantes:

- search\_header: se utiliza para buscar el nombre de un header dentro del array de estructuras *header\_line* y devolver su *header\_value* correspondiente. Esta búsqueda la realiza de manera *case-insensitive*.
- valid\_request\_info: recibe una estructura *request\_parser\_info* y devuelve true si se encuentra en la misma, una vez realizado el parseo del *request*, la dirección y el puerto necesarios para conectarse con el *origin server*. Caso contrario devuelve false.
- is\_value\_in\_header: recibe un string y el contenido de un *header\_value*, y devuelve true, si el string pasado se encuentra en el *header\_value*. La búsqueda la realiza de forma *case-insensitive*, separando el contenido del *header value* en los diferentes valores por medio del caracter separador “,”.

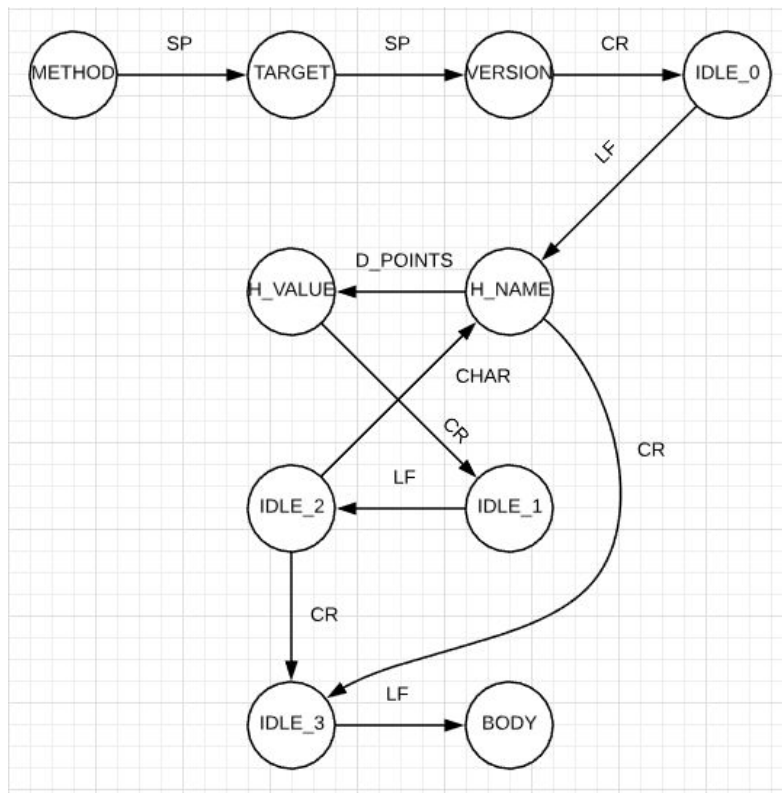
## parser

### Objetivo de parser.c:

Proveer la lógica necesaria para realizar el parseo de los mensajes HTTP. Además, se incluyen funciones que permiten inicializar las estructuras requeridas para el funcionamiento del parser, discutidas en el apartado sobre el archivo *parser\_definitions.h*.

### Funciones que abarca:

- parse\_message: realiza el análisis de los headers de un mensaje HTTP, utilizando un buffer de entrada y otro de salida en donde deja el mensaje una vez analizado. Finalmente, completa la estructura *http\_message*, pasada como parámetro, con la información parseada. Se utiliza una máquina de estados que se presenta a continuación.



- parse\_request: recibe un buffer de entrada con el contenido a parsear, y otro de salida en el que dejará el mensaje una vez analizado. Se realiza una llamada a la función *parse\_message* para el análisis de los headers. Una vez terminado esto, se analiza la

información recolectada para decidir si se cuenta con la información necesaria para conectarse con el *origin server*.

- parse\_response: su lógica es similar al de la función *parse\_request*. La diferencia reside en que, una vez que se terminan de parsear los headers, en lugar de determinar si se puede establecer la conexión con el *origin server*, se determina si el header “Content-Type” contiene un valor que esté configurado para habilitar las transformaciones. Además, una vez que se llega al estado BODY, se determina si el mismo contiene un “Content-Length” definido o si se recibió un “Transfer-Encoding: chunked”, y se realiza el parseo del mismo de manera acorde.
- parse\_chunked: realiza el parseo del BODY de una respuesta HTTP, cuando fue recibido el header “Transfer-Encoding: chunked”. Esta función utiliza una máquina de estados propia.

### Decisiones de implementación tomadas:

En caso de encontrar un error durante el parseo de los headers o del contenido del BODY (en el caso de “chunked”), se decidió optar por convertir el proxy en un “pasamanos”, copiando lo que se recibe en el buffer de entrada al buffer de salida para la función en la que se halló el error. Por ejemplo, imaginemos que en la *request* se recibió un header mal formado. Lo que sucederá, es que se dejará de analizar el contenido de los demás headers. Si esto ocurre una vez que se pudo obtener la información necesaria para conectarse con el servidor, esto ocurrirá y se le enviará el *request* al mismo, a pesar de haberse encontrado un error en el mensaje. Si esta información no se pudo obtener para el momento de hallar el error, no se establecerá la conexión con el servidor.

Por otro lado, se decidió restringir el largo de los elementos http analizados. El largo máximo con el que trabaja el proxy es de 512 bytes por elemento. Esto es para reducir la cantidad de espacio que se reserva por cliente. Asimismo, el largo máximo de un encabezado (*status-line* y *headers*) no debe superar los 4K y se almacenarán hasta 20 headers.

## store\_manager

### Objetivo de store\_manager.c:

Se encarga de manejar el almacenamiento de la información procesada por el parser. El objetivo es proveer tanto al parser como al proxy, de una estructura completa y confiable que puedan consultar para obtener toda la información que precisen. La estructura es completa porque no sólo se almacenan los elementos fundamentales (headers “Host”, “Content-Length”, etc) sino que se almacenan todos, desde el método y la versión, hasta el último header (siempre y cuando el parser no haya encontrado un error). Por otro lado, decimos que es confiable, porque la información se almacena de forma prolija (eliminando espacios en los headers, por ejemplo). De esta forma, facilita la labor de quien precise la información.

### Funciones que abarca:

- *store\_information*: recibe las estructuras *buffer* y *http\_message* analizadas anteriormente en este documento, así como el estado en el que se encuentra la máquina de estados del parser. Esta función se encarga de almacenar la información encontrada en el buffer (correspondiente a un elemento http) en la estructura *http\_message*. Para ello, utiliza funciones auxiliares.

### Decisiones de implementación tomadas:

Se decidió por almacenar toda la información posible en relación a la línea de estado (*status\_line*) y a los headers, con el objetivo de no tener que estar modificando el parsing de la información cada vez que se quiera agregar una nueva funcionalidad al proxy. Algunas de ellas, como el chequeo del header “Content-Encoding” en busca del valor “gzip”, se pudo implementar sin realizar modificaciones al parsing de los mensajes. La información necesaria ya se encontraba al alcance de proxy.

## Diagrama de secuencia

El siguiente diagrama se encuentra disponible en versión html en

<http://54.233.125.152:9090/diagrama-1.html>

**Aclaración:** Por un problema de visualización y espacio, no se adjunta la imagen del diagrama en el informe.

El diagrama de secuencia corresponde a la llamada de las funciones que se hacen al comenzar la ejecución del proxy. Además, se muestra con detalle la función `resolve` en **`clients_manager.c`** que, como se mencionó anteriormente en el informe, es la función que se encarga de “encolar” las peticiones del cliente en el selector.



## Monitoreo/Configuración

El proxy provee las opciones de configurar variables que influyen en su comportamiento y de monitorear el mismo mediante la obtención de diversas métricas. A continuación se listan las mencionadas alternativas:

- Activar o desactivar las transformaciones externas.
- Definir la lista de Media Types transformables.
- Establecer el comando a ejecutar externamente.
- Obtener las siguientes métricas:
  - Conexiones concurrentes al momento de la petición.
  - Cantidad de conexiones históricas desde el momento en que se levantó el proxy.
  - Cantidad de bytes transferidos por los clientes desde el momento en que se levantó el proxy.
  - Cantidad de bytes transferidos por los servidores desde el momento en que se levantó el proxy.

Para lograr la conexión entre el proxy y un cliente que desea hacer uso de estos servicios se utiliza un socket pasivo (socket de configuración) configurado sobre el protocolo de transporte SCTP y que escucha por defecto en el puerto 9090. Este socket es del estilo uno-a-muchos, permitiendo múltiples asociaciones activas simultáneamente, por lo que al recibir datos de un cliente con `sctp_recvmsg()`, guarda la dirección del mismo para poder enviarle la respuesta con `sctp_sendmsg()`.

## Protocolo de Transporte: SCTP

Se utilizaron los siguientes servicios provistos por el protocolo:

- Multihoming: Se configura el socket maestro de configuración para que escuche en todas las interfaces disponibles, de modo de posibilitar el acceso a la configuración y obtención de métricas desde más de una de ellas.

- Notificaciones: Se registran los eventos de conexión y desconexión para registrar el ID de cada asociación. Para posibles extensiones, hay distintos tipos de eventos que brindan información sobre las conexiones.

## Protocolo de Aplicación

Inicialmente, el servidor inicia el servicio escuchando en el puerto SCTP 9090. Cuando el cliente lo requiera, se establece una conexión mediante dicho protocolo, y el servidor asocia un ID al cliente y se mantiene la sesión mediante los siguientes estados:

- NOT\_AUTH
- AUTH
- QUIT

Primeramente, el estado es NOT\_AUTH, y se espera que el cliente envíe la contraseña de autenticación. Si la contraseña es incorrecta, se debe volver a intentar hasta que lo sea. Caso contrario (contraseña correcta), el estado pasa a ser AUTH, y el cliente puede realizar pedidos, cada uno con una respuesta correspondiente. Se sigue en este régimen de pedido-respuesta hasta que el cliente cierre o se aborte la conexión.

Frente a pedidos que no se correspondan al estado autorizado o al recibir comandos inválidos, el servidor responde apropiadamente para cada caso con un mensaje de error.

Todos los comandos y las respuestas serán de una línea, contemplando caracteres ASCII, case-sensitive y terminados en el par CRLF. Cualquier carácter enviado después no será tenido en cuenta.

Existen dos tipos de comandos:

- Sin argumentos, conformados por una letra mayúscula o número, más el par CRLF.
- Con argumentos de longitud variable (con un máximo de 128 caracteres), siendo la composición del comando comenzada por una letra mayúscula, un espacio (código decimal 32), el argumento y finalizada por el par CRLF.

Las respuestas, a su vez, serán de longitud variable, iniciadas por el código de respuesta (OK o ERROR, que se corresponden a la correcta completitud de la tarea indicada por el comando o no,

respectivamente), seguidas de un espacio, un argumento que detalla el comportamiento de la respuesta, y finalizadas en CRLF.

### Estado NOT\_AUTH:

Estado inicial de la conexión, el cliente debe enviar el comando con una contraseña correcta para poder manejar configuraciones u obtener métricas, teniendo indefinidas posibilidades de hacerlo. Si la contraseña es correcta, se pasa al estado AUTH, si es incorrecta, se permanece en este estado. Si se ingresa el comando de salida, se pasa al estado QUIT.

Posibles comandos:

- P

Argumento: algunaContraseña

Posibles respuestas: OK Authorized

ERROR Incorrect Password

Comentarios: Debe ser el primer comando para iniciar la sesión de configuración y disponer de todos los servicios. El servidor analiza la contraseña e indica si es correcta (primera respuesta) o no (segunda respuesta). La contraseña debe estar compuesta por caracteres ASCII y tener hasta 128 caracteres.

Ejemplos: C: PASS pass123

S: OK Authorized

C: PASS pw987

S: ERROR Incorrect Password

- Q

Sin argumento

Respuesta: OK Quit

Comentarios: Seguido de este comando, se procede a pasar al estado QUIT y a cerrar la conexión.

Ejemplo: C: Q  
S: OK Quit

- Comando Inválido

Respuesta: ERROR Not Authorized

Comentarios: Cualquier otra combinación, en este estado, es tenido en cuenta por el servidor como si el cliente no estuviera autenticado, haciéndoselo saber como respuesta.

Ejemplo: C: A  
S: ERROR Not Authorized

## Estado AUTH:

Una vez ingresada la contraseña válida y el servidor habiendo respondido 'OK Authorized', el cliente puede comenzar a enviar los distintos comandos para setear alguna configuración del servidor u obtener métricas. Si hubiese algún error al acceder a las métricas o variables de configuración, se lo indica con comandos de error. Notar que a los siguientes comandos se añade el previamente comentado 'Q' para finalizar la sesión.

Posibles comandos:

- A

Sin argumento

Posibles respuestas: OK Transformations Activated

ERROR Transformations Not Activated

Comentarios: Comando para activar las transformaciones externas de Media Types, que serán seteadas o reseteadas por los comandos 'T' y 'R' respectivamente.

Ejemplo: C: A

S: OK Transformations Activated

- D

Sin argumento

Posibles respuestas: OK Transformations Deactivated

ERROR Transformations Not Deactivated

Comentarios: Comando para desactivar las transformaciones externas de Media Types.

Ejemplo: C: D

S: OK Transformations Deactivated

- T

Argumento: listaDeMediaTypes

Posibles respuestas: OK Media Types Set

ERROR Media Types Not Set

Comentarios: Comando para setear los Media Types para ser transformados desde el servidor. Desde el primer caracter leído después del espacio que separa a la letra del comando hasta leer el par CRLF, se estará leyendo el argumento que corresponde a la lista de Media Types a tener en cuenta por el servidor. Para indicar la finalización de un Media Type y el comienzo de otro, se utiliza la coma (',', código decimal 44). No se analiza la correctitud de los Media Types, delegando esa tarea al cliente, ya que su no correctitud no implica un mal funcionamiento de parte del servidor. No hay un máximo especificado para la cantidad de Media Types, pero sí para la longitud de la lista completa (128 caracteres).

Ejemplo: C: T text/plain,application/x-csh,image/\*

S: OK Media Types Set

- R

Sin argumento

Posibles respuestas: OK Media Types Reset

ERROR Media Types Not Reset

Comentarios: Comando para resetear la lista de Media Types a ser transformados desde el servidor (elimina todos los Media Types de la lista).

Ejemplo: C: R

S: OK Media Types Reset

- C

Argumento: comandoAEjecutar

Posibles respuestas: OK Command Set

ERROR Command Not Set

Comentarios: Comando para setear el comando que ejecutará el sistema para aplicar la transformación externa a los Media Types preseleccionados. Desde el primer caracter leído después del espacio que separa a la letra del comando hasta CRLF se estará leyendo el argumento que corresponde al comando a tener en cuenta por el servidor. No se analiza la correctitud del comando a ser ejecutado, delegando esa tarea al cliente. La longitud máxima especificada del comando a ser ejecutado es de 128 caracteres.

Ejemplo: C: C sed -u -e s/a/4/g


S: OK Command Set

- 1

Sin argumento

Posibles respuestas: OK coneccionesConcurrentes

ERROR Metric Not Obtained



Comentarios: Pedido de cantidad de conexiones de clientes HTTP concurrentes. Se devuelve error si no se pudo obtener la métrica.

Ejemplo: C: 1

S: OK 7

- 2

Sin argumento

Posibles respuestas: OK accesosHistóricos

ERROR Metric Not Obtained

Comentarios: Pedido de cantidad de accesos históricos de clientes HTTP. Se devuelve error si no se pudo obtener la métrica.

Ejemplo: C: 2

S: OK 10

- 3

Sin argumento

Posibles respuestas: OK bytesClientes

ERROR Metric Not Obtained

Comentarios: Pedido de cantidad de Bytes transferidos por los clientes. Se devuelve error si no se pudo obtener la métrica.

Ejemplo: C: 3

S: OK 2450

- 4

Sin argumento

Posibles respuestas: OK bytesServidores

ERROR Metric Not Obtained

Comentarios: Pedido de cantidad de Bytes transferidos por los servidores. Se devuelve error si no se pudo obtener la métrica.

Ejemplo: C: 4

S: OK 4685

- ComandoInválido

Respuesta: ERROR Unknown Command

Comentarios: Cualquier otra combinación, en este estado, es tenido en cuenta por el servidor como un comando inválido.

Ejemplo: C: J

S: ERROR Unknown Command

## Estado QUIT:

Estado intermedio entre el ingreso del comando de salida tanto en el estado AUTH como en NOT\_AUTH y el cierre de sesión del cliente.

## Ejemplo de uso:

S: <esperando conexión>

C: <establecer conexión>

C: P pass123

S: OK Authorized

C: T text/html,image/\*

S: OK Media Types Set





C: A

S: OK Transformations Activated

C: 1

S: OK 2

C: 3

S: OK 126

C: Q

S: OK QUIT

C: <cerrar conexión>

S: <esperando nueva conexión>

## Problemas encontrados en general

En rasgos generales durante la realización del trabajo práctico, nos hemos encontrado con varias dificultades, enumeradas a continuación:

El primer conflicto que encontramos fue el de integrar nuestro selector, con la librería de selector brindado por la cátedra. Esto implicaba que debíamos hacer un importante refractoreo sobre lo que ya estaba armado, y eso también conllevaba a generar nuevos conflictos que previamente ya habíamos resuelto.

A si mismo, debíamos hacer la implementación de un proxy no bloqueante, y la función de resolución de nombres (mencionada anteriormente en este informe) lo es. Por lo tanto, esa función la tuvimos que llevar a un thread aparte y enviar una señal al selector. Esto debíamos solucionarlo porque era un requerimiento indispensable. Si hubiésemos usado el selector de la cátedra, no era necesario realizarlo, ya que la librería ya contempla casos como esa.

El problema con mayor impacto fue el del comando sed como función transformadora. La función transformadora se manda a un proceso hijo y envía la respuesta al cliente utilizando pipes. Podíamos observar (utilizando strace -ff) que las system calls de write y read se estaban haciendo correctamente sobre los file descriptors correctos, entonces pudimos encontrar que el problema no fue de una implementación nuestra, sino que de sed y la versión que estábamos usando para testearlo. La solución, también brindada por la cátedra, fue poner el comando -u (<https://www.gnu.org/software/sed/manual/sed.txt>), Buffer both input and output as minimally as practical. (This is particularly useful if the input is coming from the likes of 'tail -f', and you wish to see the transformed output as soon as possible.).

Una de las cosas que más nos costó es entender a fondo el movimiento de datos entre buffers. Cómo utilizamos varios buffers (ej: client\_read, client\_write, server\_read, etc) y tambien el parser tiene varios funcionamientos que los utilizan (parse\_request, parse\_response, transformacion, etc) se complica bastante el flujo de datos entre estos e incluso cuando hay que mover los datos con respecto a las syscalls.

En cuanto al parseo de los datos, el principal conflicto es cuando el contenido llega zippeado y chunkeado. Su solución, si bien fue implementada rápida, el entendimiento del conflicto fue lo que nos demoró.



## Posibles extensiones

Debido a las estructuras que manejamos, una posible extensión sería de agregar una función extra que sea, no solo filtrar por Media Types para las transformaciones, sino que, cuando recibe del cliente el hostname o ip, podamos filtrar las conexiones a origin server.