Protocolos de información - 2019 1C

Servidor HTTP

Integrantes

Nombre	*Correo*	*Legajo*
Dammiano, Agustín	adammiano@itba.edu.ar	57702
Donoso Naumczuk, Alan	adonoso@itba.edu.ar	57583
Izaguirre, Agustín	aizaguirre@itba.edu.ar	57774
Sanz Gorostiaga, Lucas	lsanz@itba.edu.ar	56312

Fecha: 18/06/2019

1. Índice

2. Descripción del protocolo y las aplicaciones desarrolladas

Se pide implementar un servidor proxy de HTTP que pueda funcionar de forma transparente por los User Agents. También, fue pedido realizar un aplicación para poder manejar dicho servidor. Además, se diseñó un protocolo para poder comunicar ambos programas. Este sección va ampliar sobre los mismos.

2.1. Proxy HTTP

En este trabajo se implementó un servidor proxy para el protocolo HTTP/1.1 que no soporta conexiones persistentes (por cada request se establece una nueva conexión). Es un servidor concurrente con entrada y salida no bloqueante multiplexada.

Para el desarrollo del trabajo práctico se utilizaron los siguientes códigos fuentes provistos por la cátedra smt.c, smt.h, buffer.c, buffer.h, selector.c y selector.h, de los cuales solo se modificó buffer.c para agregarle un nuevo puntero que será discutido en la sección de problemas encontrados.

Para los requests y response todas las líneas deberían terminar CRLF, pero también soportamos si son solo terminadas en LF. Los métodos soportados son GET, HEAD, PUT, DELETE y POST.

2.1.1. Estados del proxy

Son los estados por los que irá atravesando el proxy durante su ejecución (ver figura 1).

2.1.1.1. PARSE

Descripción

En este estado se parsea el principio de la request del cliente hasta encontrar el host ya sea en la primer línea o en un un header. En caso de no encontrar el host en la primera línea se examinarán los headers del request en busca del header host poniendo una cota en la cantidad de bytes leídos (1000).

Se transiciona al estado de CLIENT_ERROR en caso de que suceda un error que requiera mandar un mensaje al cliente, en caso contrario llega a CONNECT_TO_ORIGIN.

Primera línea del request

Al llegar una request al proxy la primer línea es parseada. Primero para registrar el tipo de operación que se quiere realizar (GET, HEAD, DELETE, PUT o POST), con el fin de saber si dicha operación es soportada por el proxy (ver figura 2). Después se parsea el target con el fin de encontrar un host o puerto (ver figura 3). Finalmente, se obtiene la versión del protocolo con el fin de corroborar que un versión aceptada (ver figura 4), 1.1 o 1.0.

Parseo del header host

Para conectarse al servidor de origen y poder realizar la petición es necesario obtener la dirección IP del origen, la cual se puede obtener al realizar una query de DNS del host del origen. Este host, como se mencionó previamente, se puede obtener del target en la primer línea del request HTTP pero en algunos casos es imposible. Por esto, si no se pudo encontrar en la primer línea, el programa sigue parseando los header con el fin de encontrar el host (ver figura 5). Tanto la primer línea como todo lo leído hasta encontrar el host es guardado en un buffer con el fin de poder mandárselo al origen cuando se establezca la conexión.

2.1.1.2. CONNECT_TO_ORIGIN

Descripción

En este estado se resuelve la IP del host obtenido en caso de ser necesario y se intenta conectar al mismo.

Si logra conectarse se transiciona al estado de HANDLE_REQUEST, mientras que si sucede un error con la resolución del nombre o con la conexión se transiciona al estado ERROR_CLIENT y en caso de error interno al estado ERROR.

Resolución del host

Después de haber conseguido el host es necesario realizar una query de DNS para obtener la IP. Pero para realizar la misma es necesario bloquearse y esperar la respuesta. Como no se quería bloquear al servidor (debido que este iba a perder tiempo que podría estar usando para atender a otros cliente) se optó por realizar un thread que realice la consulta. El thread principal saca los intereses del selector y cuando el thread secundario terminaba este lo vuelve a registrar. De esta manera el thread principal tiene cargado la lista de resoluciones (ya que los threads comparten heap), y puede iterar sobre la misma para intentar conectarse. En caso de no poder la aplicación devuelve un error de la familia 500 al cliente.

2.1.1.3. HANDLE_REQUEST

Descripción

En este estado se lee la request del cliente hasta que reciba una respuesta del origin server, en dicho caso transiciona al estado HANDLE_RESPONSE, en caso de cualquier error transiciona al estado ERROR.

Tanto en este estado como en HANDLE_RESPONSE se utiliza un parser(nos referiremos a este parser como headersParser) para filtrar los headers tanto del request como del response.

Comportamiento del headersParser

Tiene una máquina de estados(Ver figura 6), recibe un buffer y va dejando el resultado de lo que parseo en un buffer(nos referiremos a él como parsedBuffer).

Primero se guarda la primer línea del request(todo lo leído hasta el primer LF), soporta tanto el uso de CRLF como el de LF, para luego ser logueada.

Dentro del estado de HEADERS_NAME va guardando el nombre del host en un arreglo de tamaño constante, donde el tamaño fue elegido de acuerdo a la longitud del header mas largo que queríamos filtrar ya que si un header tiene longitud mayor a esa ya sabemos que no lo censuramos. Los headers se guardan en minúscula ya que son case-insensitive(según el RFC 2616).

Dentro del estado de HEADER_VALUE escribimos directamente en el parsedBuffer si es un header que no censuramos o no escribimos nada si es un header que censuramos, también dependiendo del header name leído guardamos el valor. Pero esto será explicado en HANDLE_RSPONSE ya que la razón de guardar los valores está relacionada con el estado HANDLE_RESPONSE.

Los headers censurados del request son keep-alive, connection, upgrade, trailer(solo si están activadas las transformaciones). Los headers connection y keep-alive son censurados ya que no permitimos conexiones persistentes y por ese motivo también al final de los request headers agregamos el header connection: close.

Lectura, Filtrado y Envío de headers del request

Se usan tres buffers distintos en este estado, el buffer donde se leyó la primer parte del request hasta encontrar al host(nos referiremos a este buffer como requestBuffer), el buffer de donde se seguirá leyendo del cliente(nos referiremos a este buffer como clientBuffer) y el buffer
del cual enviamos la información al origin server(nos referiremos a este buffer como parsedBuffer).

Primero se pasa todo lo leído por el requestBuffer por la máquina de estados del parseHeaders, al atravesar la misma, quedará en el parsedBuffer la información que queremos enviar al origin server. También se guarda la primer línea de la respuesta para luego ser utilizada por el logger. Se comparan también dentro de esa máquina de estados si están activadas las transformaciones los valores de content-encoding, transfer-encoding ambos con sus valores en minúscula(ya que son case insensitive según RFC 2616) ya que su valor influencia el comportamiento del estado TRANSFORM_BODY. También se guarda el valor del header content-type para luego compararlo con el filtro establecido y decidir si se efectúa o no la transformación.

Finalmente escribimos del parsedBuffer al origin, este ciclo se repite hasta recibir una respuesta del origin server. Donde se transiciona al estado HANDLE_RESPONSE.

2.1.1.4. HANDLE_RESPONSE

Descripción

Este estado tiene dos comportamientos, uno cuando están activadas las transformaciones y otro cuando no están activadas las mismas.

En ambos casos parsea los headers de la respuesta, si están activadas las transformaciones al terminar los headers transiciona al estado TRANSFORM_BODY, si no están activadas todo lo que lea del Body del origin lo pasa al cliente como lo recibió y si recibe algo del cliente también se lo manda directamente como lo recibió.

Lectura, Filtrado y Envío de headers del response

Se usan dos buffers distintos, del que lee la respuesta del origin(nos referiremos a este buffer como responseBuffer) y el buffer donde deja la información que queremos enviar al cliente(nos referiremos a este buffer como parsedBuffer). Ademas de los headers sensurados en HANDLE_REQUEST en este estado se sensuran si estan activadas las transformaciones los headers transfer-encoding y content-length.

Primero, se pasa todo lo leído del responseBuffer al headersParser(detallado en HANDLE_REQUEST), el parser en caso de tratarse de una respuesta guarda los valores para los headers content-type(para luego aplicarle el filtro de mime-types y decidir si se efectúara la transformación), transfer-encoding(para ver si la respuesta será chunkeada) y content-encoding (ya que si el valor de este header no es el de identity no se realizará la transformación).

Luego se envía esa información al cliente. Como se dijo en la descripción, si no están activadas las transformaciones(por defecto al introducir un comando junto con la ejecución del proxy se activan, luego pueden prenderse y apagarse con el administrador) sigue pasando el body del response como lo recibe.

2.1.1.5. TRANSFORM_BODY

Descripción

Este estado tiene cuatro comportamientos distinto dependiendo del estado interno del proxy con el que llega a este estado, siempre que se llegue a este estado las transformaciones van a estar activas. En este estado se envia el body de la respuesta del origin al cliente pudiendo aplicarle antes transformaciones.

Comportamientos

También en este estado se usa un parser para deschunkear la información, solo para el comportamiento 4 (ver figura 7).

En este estado en total se utilizan cuatro buffers, uno del que se lee del origin(nos referiremos a este buffer como originBuffer), uno del que se lee del comando transformador(nos referiremos a este buffer como transformBuffer), uno en el que se chunkea la información leída del comando transformador o del origen, dependiendo el comportamiento(nos referiremos a este buffer como chunkBuffer) y uno donde se guarda la informacion leida del origen deschunkeada cuando el origen envía su respuesta chunkeada(nos referiremos a este buffer como unchunkBuffer)

1. Cuando el content-type no coincide con los establecidos en el filtro para transformar o cuando el programa transformador falla o cuando el content-encoding es distinto de identity, en este caso enviará todo lo que lea del origen como lo recibe pero chunkeando la información. Se utilizan únicamente el originBuffer y el chunkBuffer.
2. Cuando el content-type coincide con los establecidos en el filtro, el comando es ejecutado correctamente y no hay content-encoding o su valor es identity. En este caso lee la respuesta del origen y se la envía como la recibe al programa

transformador, del cual luego lee y chunkea la información obtenida para enviarla al cliente. Se utilizan el originBuffer, transformBuffer y chunkBuffer.

3. Cuando el content-type no coincide con el del filtro y la respuesta del origin viene chunkeada, en este caso no se chunkea la información leída del origen sino que se envía igual a como es recibida. Se utiliza únicamente el originBuffer.
4. Cuando el content-type coincide con el filtro, el comando es ejecutado y no hay content-encoding o su valor es identity. En este caso lee la respuesta de origen la deschunkea y se la pasa al comando transformador, luego lee la información del comando transformador y la chunkea para luego enviarla al cliente. Se utiliza el originBuffer, el unchunkBuffer, el transformBuffer y el chunkBuffer.

2.1.1.6. DONE

Descripción

Es al estado que se llega cuando se terminó de enviar la respuesta del origin server al cliente. En este estado se cierra la conexión y se liberan los recursos almacenados para esta coneccion.

2.1.1.7. ERROR

Descripción

Es el estado al que se llega cuando ocurre algún error independiente del cliente o del origin server, se cierra la conexión y se liberan los recursos almacenados.

2.2. Sistema de Logging

El sistema de logging implementado consta de tres niveles: *access*, *error* y *debug*. El primero mantiene un registro de los pedidos y respuestas al proxy guardando en un archivo *logger/logs/access.log* cada pedido/respuesta en una línea con el siguiente formato:
`[fecha_y_hora] [ip_cliente --> ip_destino] - 'request_o_response'.`

El log de errores se utiliza para mantener un registro de errores que hayan ocurrido en el código del proxy. Se encuentra en *logger/logs/error.log* y su formato es:
`[fecha_y_hora] descripción_custom_del_error: descripción_de_errno o`
`[fecha_y_hora] Error: descripción_de_errno`

El tercer nivel es de utilidad para desarrollar, no debería ser utilizado en circunstancias de ejecución normal. Por otro lado, los dos primeros niveles siempre se encuentran activos.

2.3. Métricas

Se implementó un sistemas de métricas para el monitoreo del proxy. Las mismas constan de cantidad de bytes transferidos, conexiones concurrentes y conexiones históricas. La primer describe los bytes que transfiere el proxy. La segunda cuenta la cantidad de

clientes están conectados al proxy (contando los administradores). La última contabiliza la cantidad total de accesos al proxy. Estas pueden obtenerse mediante el administrador.

2.4. Protocolo desarrollado

2.4.1. Introducción

Se desarrolló un protocolo binario para obtener y editar recursos. El protocolo soporta versionado, autenticación y tres operaciones: BYE, GET y SET. Está diseñado para una arquitectura de cliente-servidor (múltiples clientes concurrentemente), en la que el servidor se encuentra esperando que se efectúe la etapa de autenticación, iniciando siempre la comunicación el cliente, y una vez haya sido completada correctamente, dicho cliente obtenga permisos para operar mediante los comandos GET (obteniendo recursos) y SET (modificando recursos) ciertos recursos que se encuentren en el servidor.
Una vez finalizada la interacción, el cliente envía el comando BYE para indicar que no solicitará nada más y cierra su conexión.

A continuación se detalla en formato ABNF (RFC 5234) las distintas solicitudes (o *requests* a partir de ahora) y respuestas (o *responses* a partir de ahora) definidas en el protocolo.

2.4.2. Autenticación

Como se mencionó anteriormente, el cliente es quien inicia la comunicación. Lo hace mediante una solicitud de autenticación (**`authentication-request`**), la cual tiene la siguiente estructura:

```
authentication-request = version username password


version = version-representation-bit-qty
          separator
          version-value
          [fill-to-octet-multiple]

version-representation-bit-qty   = "1"
separator                        = "0"
version-value                    = "0"
fill-to-octet-multiple           = 1*BIT

username = 1*CHAR %x00
password = 1*CHAR %x00
```

Como se puede ver, cuenta con una cabecera de **`version`**, que a tiene tantos bits encendidos como el mínimo número de bits que se necesiten para representar la versión del protocolo en número binario (en el ABNF presentado, se definió la estructura de versión para

la versión que corresponde, en este caso la cero), seguido de un bit apagado que actúa como separador y luego la representación binaria del número de versión del protocolo utilizando la mínima cantidad de bits posibles. Si la cantidad de bits utilizados para la estructura de versionado que se detalló no es múltiplo de 8, se agregan a la derecha los bits necesarios para que lo sea; estos bits serán ignorados y su única función es alinear a byte (8 bits) el inicio de los próximos datos: el usuario y la contraseña, ambos cadenas de caracteres terminados con el carácter nulo.

Luego, el servidor responderá con una respuesta de autenticación (**authentication-response**), la cual se detalla a continuación:

```
authentication-response = general-status
                          version-status
                          authentication-status
                          [version / 5BIT]

general-status        = ok / error
version-status        = ok / error
authentication-status = ok / error

ok    = "0"
error = "1"
```

Esta cuenta con una cabecera de 3 bits, que se utilizan para indicar el estados de éxito o error. El primer bit indica el estado general de la respuesta, luego le siguen el bit que indica el estado de la versión, siendo 1 si el servidor está utilizando una versión distinta o incompatible a la del cliente y el bit de estado de autenticación que indica si el usuario y contraseña enviados previamente en la solicitud son correctos.

En caso de que el estado de versión esté en estado de error, a continuación de la cabecera de 3 bits seguirá una estructura de versión como la que se vió en la descripción de la solicitud de autenticación. Si el estado de la versión es exitoso, se rellenará a derecha con tantos bits como sean necesarios para alcanzar un múltiplo de 8 bits, en este caso como se trata de la versión cero, se rellena con 5 bits.

Es importante tener en cuenta que si bien el protocolo ofrece versionado, las distintas versiones no deberían modificar la fase de autenticación que se describe en la versión cero, ya que no podrían comunicar los datos esenciales anteriormente nombrados. Por el contrario, de necesitar modificar la fase de autenticación, se espera que luego de la etapa de autenticación descrita, agreguen nuevas etapas que sólo sean comprensibles por nuevas versiones.

2.4.3. Solicitudes operacionales

Como se mencionó, se cuenta con tres operadores (BYE, GET y SET). Por cuestiones de seguridad y eficiencia, el protocolo define a las operaciones GET y SET como operaciones condicionales, esto se detalla próximamente.

Las solicitudes de operación tienen la siguiente estructura:

```
request = opcode resource-id [time-tag] [data]

resource-id = 8BIT
time-tag    = 64BIT

opcode = bye / get / set

bye     = "00"
get     = "01"
set     = "10"
```

Cuentan con un byte de cabecera, que cuenta con los primeros dos bits para indicar operador y los siguientes 6 bits para indicar el recurso sobre el cual operar. Además, el GET y el SET requieren **time-tag** (se detalla a continuación) y el SET requiere **data**.

2.4.3.1. Operación BYE

Esta operación se utiliza para indicar el fin de la comunicación por parte del cliente, sirve en caso de que el servidor necesite realizar alguna acción al finalizar la conexión y/o se quiera evitar que todos los cierres de conexión actúan como cierres de comunicación repentinos.

Particularmente, un request para esta operación sería:

```
bye-request = bye 6BIT
```

Como podemos ver, los primeros dos bits indican el operador BYE, por lo cual serian ambos cero y luego vendría el identificador de recurso, pero como el BYE no opera sobre ningún recurso, se rellena con 6 bits a la derecha, para que sea múltiplo de 8 bits.

Esta operación no tiene definida una respuesta, pero podría definirse por el usuario del protocolo de ser necesario.

2.4.3.2. Operación GET

Esta operación se diseñó con el fin de que un cliente obtenga un recurso alojado en el servidor. Cuenta con la siguiente estructura:

```
get-request = get resource-id time-tag
```

Como se espera, los primeros dos bits indican el operador GET, siendo 0 y 1 respectivamente, seguidos de 6 bits indicando el identificador del recurso a obtener.

Como se mencionó anteriormente, es un operador condicional y aquí es donde entra en juego el **time-tag**, una cadena de 64 bits cuyo objetivo es representar la fecha y hora de la última modificación del recurso. De esta forma, el servidor sólo nos devolverá el recurso

solicitado si tiene alojada una versión con **time-tag** distinto al que enviamos en la solicitud, haciendo eficiente la solicitud de datos de gran tamaño.

2.4.3.3. Operación SET

Esta operación se diseñó con el fin de que un cliente modifique un recurso alojado en el servidor. Cuenta con la siguiente estructura:

```
set-request = set resource-id time-tag data
```

Los primeros dos bits como 1 y 0 representando el operador SET, seguidos de 6 bits representando el recurso a modificar.

Como se mencionó, el SET es un operador condicional, se diseñó esperando que el servidor sólo modifique el recurso si el **time-tag** de la request coincide con el **time-tag** de la última versión que se aloja en el servidor, evitando que se pueda modificar un recurso desconociendo su estado actual en el servidor.

Por último, la **data** será el nuevo valor del recurso. Se detalla más adelante de qué manera se envía la **data**.

2.4.4. Respuestas operacionales

Las respuestas a las operaciones de GET y SET tienen la siguiente estructura:

```
response = general-status
           opcode-status
           id-status
           time-tag-status
           4BIT
           [time-tag]
           [data]

general-status  = ok / error
opcode-status   = ok / error
id-status       = ok / error
time-tag-status = ok / error
```

Cuentan con una pequeña cabecera de 4 bits, en la cual se indica el estado general de error, la validez del operador (en la versión cero del protocolo, podemos ver que el operador representado por la cadena de bits 11, no está definido, un usuario del protocolo podría utilizar dicho operador definiéndolo a gusto, o bien podría setear el **opcode-status** en error), el estado del identificador de recurso (podría utilizarse menos recursos que la cantidad que puede ser representada por 6 bits, haciendo que ciertas cadenas de 6 bits sean inválidas como identificador) y el estado del **time-tag** que indicará error en caso de que el enviado en la solicitud no coincida con el **time-tag** que tiene el servidor sobre ese recurso.

A continuacion 4 bits de relleno, para alinear a byte y dependiendo del los valores de los bits de estado y de la operación solicitada, podría enviarse un **time-tag** y/o datos.

2.4.4.1. Operación GET

Como se detalló anteriormente, el envío de **time-tag** y/o **data** depende de la solicitud y de los bits de status. Para el caso del GET, sólo devolverá **time-tag** y **data** en caso de que el **time-tag** enviado en la solicitud no coincida con el que se tiene en el servidor, lo cual indicaría que el recurso que tiene el cliente está desactualizado con respecto al servidor, y por tanto se devuelve el **time-tag-status** en 1 (**error**), se envía el **time-tag** actualizado con la versión que se tiene en el servidor y en la **data** se envía el recurso actualizado. En caso de que el **time-tag** coincida, se devuelve el **time-tag-status** en 0 (**ok**) y no se envia ni **time-tag** ni **data**.

2.4.4.2. Operación SET

En este caso, se diseñó pensando en que no se devolverá **data**, simplemente se devolverá **time-tag-status** en 0 (**ok**) si coinciden los **time-tag**'s y por tanto se retornará el nuevo **time-tag** que se generó al modificar el recurso mediante la operación SET solicitada, o **time-tag-status** en 1 (**error**) en caso de no coincidir los **time-tag**'s y por tanto no se devolverá ningún **time-tag**, esperando que para obtener el **time-tag** actualizado se realice una operación de GET sobre ese mismo recurso, para luego de ver el estado actual del recurso se decida si realizar la operación de SET nuevamente.

2.4.5. Datos dinámicos

No se quiso limitar el envío de datos a un tamaño particular, por lo que se diseñó un protocolo de formateo de los datos para poder enviar datos de tamaño arbitrario.

Entonces, se define la **data** de la siguiente forma:

```
data = *common-data-block final-data-block

final-data-block = is-final-byte
                   [fill-to-k-bytes-of-data]
                   start-data-byte
                   1*(k - n)concret-data

common-data-block = %x00
                    [fill-to-k-bytes-of-data]
                    start-data-byte
                    1*(k - n)concret-data

start-data-byte         = %x80
fill-to-k-bytes-of-data = 1*%x00
is-final-byte           = %x10
```

concret-data = OCTET

Se envían los datos formatearlos en bloques de datos contiguos. Cada bloque tiene un número **k** de datos concretos y **2** bytes de información del formateo, por lo tanto, cada bloque es de **k + 2** bytes (en la versión cero, se utiliza **k = 8**).

El formato consta de dividir los datos a enviar en bloques de tamaño **k** y luego, agregar en cada bloque **2** bytes, uno para indicar dónde empiezan los datos concretos de dicho bloque (el bloque podría contener **k** o menos bytes datos concretos), al cual llamamos **start-data-byte** y tiene un valor específico de **0x80** y otro para indicar si este bloque es el último bloque de la cadena de bloques contiguos de datos formateados.

Por lo tanto, cada bloque inicia con un byte indicando si es el último bloque o no, seguido de bytes de relleno con valor cero, luego el byte con valor **0x80** que indicando que el próximo byte es de datos concretos, y luego **(k** - bytes de relleno de ese bloque) de datos concretos. Idealmente, se espera que todo el relleno se encuentre en el primer bloque, y los siguientes sean de únicamente datos concretos.

A continuación podemos ver un ejemplo de una respuesta a una solicitud de GET sobre un recurso cuya representación es el string **"mycommand"**, por simplicidad se utilizo **k**, la cantidad de datos concretos por bloque, igual a **4**. Podemos ver que se responde con todos los bits de estado en **ok** excepto el primero y el cuarto, indicando que el **time-tag** no coincidió. Luego cuatro bits de relleno, 64 bits para el **time-tag** de la versión actual del recurso y luego comienza la **data**, **"mycommand"** consta de 9 bytes, por lo tanto como nuestro **k = 4** necesitamos 3 bloques y 3 bytes de relleno.

El primer bloque tiene el primer byte en **0x00** indicando que es un bloque común (es decir, no es el último bloque), seguido de los 4 bytes de relleno con valor **0x00**, seguido del **start-data-byte** indicando que a continuación vienen los datos concretos del bloque, y por lo tanto, se ve que a continuación viene el byte **0x6D** representando el valor de **'m'**.

Luego, otro bloque común, pero en este caso sin relleno. Y luego, el bloque final, que a diferencia del común, el primer byte tiene el valor **0x10** indicando ser el último.

```
Get response
(With data blocks of 4 bytes, data = mycommand)
 1 0 0 1 1111 0x000x000x000x000x5C0xF00x180xC2
 status  fill      time-tag: time command

0x00 0x000x000x00 0x80 0x6D 0x00 0x80 0x790x630x6F0x6D
com   fill zeros  sdb  'm'  com  sdb  'y' 'c' 'o' 'm'
0x10 0x80 0x6D0x610x6E0x64
fin  sdb  'm' 'a' 'n' 'd'
```

2.5. Administrador

Se implemento un programa cliente administrador para obtener métricas sobre el servidor, y modificar comando transformador, estado de transformaciones y los media ranges a transformar.

2.5.1. Utilización del protocolo para el administrador del proxy HTTP

Se implementó el protocolo descrito anteriormente utilizando el lenguaje C. Para enviar y recibir las requests y responses se utilizó el protocolo SCTP.

Se tiene la siguiente interfaz de comandos para terminal:

Obtiene el comando transformador
get cmd
Obtiene el media range
get mime
Obtiene el estado de las transformaciones (activado o desactivado)
get tf
Obtiene la cantidad de conexiones concurrentes
get mtr cn
Obtiene la cantidad de conexiones históricas
get mtr hs
Obtiene la cantidad de bytes transferido por el proxy
get mtr bt
Cambia el comando transformador
set cmd *command*
Agrega *media-range* a la lista de media-ranges del servidor
set mime *media-range*
Resetea dejando vacía la lista de media-ranges del servidor
set mime
Activa o desactiva las transformaciones dependiendo si se usa el on o el off
set tf on/off
Envía un request de BYE al servidor
bye

El administrador soporta pipelining, luego de cada comando se espera **'\n'** y para terminar la lista de comandos a procesar, se manda **'.'** seguido de **'\n'**.

Se restringe los identificadores de recurso a:

mime-id = "000001"

```
cmd-id     = "000010"
mtr-cn-id  = "000011"
mtr-hs-id  = "000100"
mtr-by-id  = "000101"
tf-id      = "000110"
```

Para **set/get cmd *command*** y **set/get mime *mime-type*** como datos concretos se envía una cadena de char null-terminated. Para **set mime** se envía el string vacío (un único byte con valor **0x00**).

Para **get mtr cn, get mtr hs** y **get mtr bt** se manda un entero de 64 bits (representado por un **uint64_t**). Se pasa a de host-bytes a big-endian antes de enviarlo y se pasa de big-endian a host-bytes al recibirlo, mediante las funciones de **be64toh** y **htobe64** de la librería **<endian.h>.**

Para **get tf** y **set tf on/off** se envía un byte **0x01** indicando **ON** y un byte **0x00** indicando **OFF**.

Para el **time-tag** se utilizaron los 64 bits que devuelve el comando **time** de la librería **<time.h>.** El servidor inicializa todos los **time-tag**'s con el valor que le devuelve el comando **time** al iniciarse. Y el cliente inicia todos los **time-tag**'s en cero.

Dado que se implementó pipelining y se implementó el protocolo sobre SCTP, se aprovechó los streams de SCTP. Se utilizaron dos streams. Uno especial para los SET y otro para los GET. De esta forma, sabremos que los SET del pipelining, se enviarán y recibirán en orden entre si, y los GET se enviarán y recibirán en orden entre si.

Para el GET se utilizó el stream número 0, para el SET se utilizó el stream número 1, para el BYE el stream número 1 y para autenticación el stream número 0.

En pipelining, se envía todos los datos sobre una mismo estado del cliente. Por lo tanto, múltiples SET en pipelining en caso de coincidir **time-tag** solo resultara exitoso el primero. Y hacer un GET y un SET en pipelining, no haría que el SET sea exitoso.

3. Problemas encontrados durante el diseño y la implementación

3.1. Proxy HTTP

3.1.1. Modificación de los headers

Cuando se implementó el proxy se tuvo que contemplar el manejo de los header HOP-BY-HOP, los cuales se permiten cambiar cuando pasan por un proxy. Primero se optó por censura el header de UPGRADE debido que el mismo puede generar que se cambie el protocolo HTTP por otro. Como el cuerpo de la respuesta puede variar por medio de las transformaciones se decidió censura el header de TRAILER si las mismas están encendidas. Esto se debe a que este header permite mandar un header en el body y entonces el cliente podría intentar buscarlo cuando en realidad este ya no existe. También, se agrega o pisa (en caso de estar) el header de TRANSFER-ENCODING con el valor de chunked en la respuesta cuando hay transformaciones. Así de esta manera se puede calcular la longitud de una parte y mandarla, no es necesario esperar a la respuesta completa del origen. Por este mismo motivo se optó por censurar el header de CONTENT-LENGTH. Como se quiso no manejar conexiones persistentes, debido a que las mismas son más complejas, se censura el header KEEP-ALIVE y se pisa o agrega un CONNECTION: CLOSE. El resto de headers no se ve modificado por el proxy HTTP.

3.1.2. Ejecución del programa transformador

Para ejecutar el comando transformador recibido compatible con system (3) usamos sh y le pasamos como argumento el comando a ejecutar.

De esta manera era complicado saber si un comando era ejecutado correctamente. Como primera medida hicimos un waitpid con el flag de WNOHANG (para que no bloquee el proceso) luego de ejecutar el comando.

Sin embargo muchas veces pasaba que al hacer el waitpid en ese momento se había ejecutado el sh pero todavía no había fallado el comando, por lo que eso no servía y al querer escribirle luego, el programa ya había terminado y daba EPIPE.

Por eso último ignoramos la señal SIGPIPE y si es recibida antes de enviarle nada al programa transformador, se manda todo el mensaje sin transformar del origen al cliente, solamente chunkeando la información.

Para saber si fallaba una segunda medida fue hacer un proceso intermedio. El proxy generaba un proceso hijo con el que se comunicaba a través de un pipe y ponía todos los intereses salvo el del file descriptor de lectura del pipe en OP_NOOP(para no ser despertado salvo por el hijo) este proceso intermedio generaba otro proceso hijo y en él ejecutaba el proceso transformador y ejecutaba waitpid con WNOHANG y luego sleep por 1 segundo, esto lo hacia 5 veces o hasta que algún waitpid le diga que termino su hijo. En el caso de que ningún waitpid le informará la finalización del hijo asumimos(falsamente) que el programa se había ejecutado correctamente y mandaba un 0 por el pipe al proceso padre, si en cambio algún waitpid indicaba que el hijo había terminado asumimos(también falsamente) que el

comando era erróneo y mandaba un 1 al padre. Por último el padre al despertarse sabía si el programa transformador había sido ejecutado o no y de acuerdo a eso elegía su comportamiento y los intereses para el selector. Este segundo método tenía muchos inconvenientes, primero si el comando transformador era ejecutado correctamente se demoraba la conexión 5 segundos que el hijo estaba esperando a ver si terminaba, en segundo lugar se asume que un comando que no falló en 5 segundos no iba a fallar mas adelante lo cual es falso. Por último también consideraba a programas correctos que terminaban antes de 5 segundos (como el ls) como inválidos.

Por eso se decidió optar por el primer método consultando una única vez si el hijo terminó con un waitpid no bloqueante y luego la primera vez que se mandaba algo si fallaba por SIGPIPE se ignoraba la señal y se enviaba el mensaje sin transformar. Pero si ya se había enviado algo y luego fallaba se enviaba un body vacío.

Para tratar de superar el problema del body vacío en caso de que falle el programa luego de mandarle algo se agregó a la implementación de buffer.c y buffer.h un puntero de progreso que se avanzaba de acuerdo a lo que se enviaba al programa transformador para no mover el puntero a read(por si después fallaba para poder mandar el mensaje desde el principio sin transformar al cliente). Sin embargo el agregar este puntero de progreso solo se solucionaba el caso de que el programa terminará antes de que se manden tantos bytes como el tamaño del buffer, si ya se había mandado un buffer completo se seguía perdiendo el mensaje y no tenía sentido. Por eso no se utilizó esta mejora en la implementación final, pero se mantuvo en el código de buffer.c por si era útil en algún otro lado.

3.1.3. Finalización del programa transformador

Tuvimos como problema que en las transformaciones largas la última parte de la transformación no se enviaba, esto se debía a que cuando terminaba la respuesta del origin, escribíamos al transform y si no había nada para leer del mismo terminamos la conexión. Entonces muchas veces se perdía la última parte de la conexión.

Para solucionar este problema se agregó un flag para cuando la respuesta del origen terminaba y si no habia nada mas para mandarle al programa transformador se le cerraba el pipe, tambien se agrego un flag para cuando el programa transformador terminaba(se leía EOF)

y una vez escrito todo al cliente si había terminado la respuesta del programa transformador se cerraba la conexión. Con esto se soluciono el problema y se pudo enviar toda la información.

3.1.4. Problema con último chunked

Cuando se hacía un request con curl y el proxy chunkeaba la respuesta del origin al cliente, se enviaba todo el mensaje pero curl devolvía un error: curl (18) transfer closed with outstanding read data remaining.

El error era que luego del último chunk(0 CRLF) no se mandaba CRLF como especifica el RFC 2616 sección 3.6.1

3.1.5. PUT y POST grandes

En el momento que se llegó a una version estable del proxy HTTP se decidió correr los casos de pruebas. Dos de esto consta en realizar un PUT y un POST grande mediante curl. Como curl manda en la request que es mayor a una cantidad X de bytes (lo cual era el caso) agrega un header EXPECT: 100-CONTINUE, con el fin de que el servidor le diga si por los header la operación va a hacer un éxito y luego ver de mandar el archivo. Pero como lo destacamos en una de la secciones anteriores modificamos varios header, esto sucede mediante un parser que ignora la primer línea del request o response. En este caso fallaba porque además de la respuesta normal de la response se agregaba otra línea antes de codigo 100, lo cual no fue contemplado.

Como primera solución se planteó censurar el header EXPECT. Esta no duró mucho, debido a que después de implementarla y probarla se noto que curl mandaba una línea diciendo que se cansaba de esperar el mensaje de código 100, la cual se mezclaba con el cuerpo del request y hacía que se corrompe el archivo subido. Después de este intento se optó por hacer que el parser interno se salte otra línea si veía el codigo 100, lo cual soluciono el problema.

3.2. Sistema de Logging

La primera dificultad para el sistema de logging fue determinar cuáles eran las necesidades del programa y definir que únicamente eran necesarios dos niveles, junto con un nivel de debugging para facilitar el desarrollo. Luego se tuvo que modificar la utilización de la función *fopen* por *open* para poder utilizar el flag *O_NONBLOCK* y así evitar que el logging fuera bloqueante.

3.3. Administrador

Al diseñar el protocolo, no se tuvo en cuenta que en el cliente se querría conocer el identificador del recurso y la operación, al recibir una respuesta (el response no incluye estos datos). Por lo tanto, en el cliente se tuvo que mantener una cola de requests enviados, una para cada stream, donde contiene la información de recurso y operación.

4. Limitaciones de la aplicación

Una limitación es que el selector por su implementación está limitado a un máximo de 1024 file descriptors al mismo tiempo y en una conexión no recursiva hay como máximo 4 file descriptors (1 por el cliente, 1 por el origin y 1 por cada punta del pipe con el comando transformador). A esto se le acumulan los files descriptors utilizados por el manager para atender sus conexiones.

A partir de esto podemos decir que si no hay conexiones con el administrador y todas las response serán transformadas puede haber un máximo de 256(1024 / 4) conexiones

concurrentes soportadas y a partir de ahí las nuevas conexiones serán rechazadas. Aunque en realidad este número es inalcanzable en ese caso debido a que no tenemos en cuenta que tenemos dos file descriptors registrados siempre(el que escucha conexiones del proxy y el que escucha conexiones del admin) por lo que el número máximo de conexiones con transformaciones seria 255(1022 / 4) conexiones con transformación y la conexión 256 no podría transformar.

Otra limitación es que las conexiones solo son terminadas cuando una de las puntas termina la conexión o ocurre un error, pero no son liberadas si al pasar un cierto tiempo no tienen interacción. Esto hace que sea más fácil de provocar un DoS(Denial of Service).

5. Posibles extensiones

Como se mencionó en las limitaciones del proxy la cantidad de conexiones está limitada, para poder solucionar ese problema se podría cambiar la implementación del selector para que utilice poll.en lugar de pselect.

Tambien podria utilizarse la alarma del selector para limpiar las conexiones que no se utilizaron en determinado tiempo.

Se pueden agregar más métodos soportados por el proxy siempre teniendo en cuenta el comportamiento esperado del proxy de acuerdo al método.

Se puede hacer que se acepten conexiones persistentes con pipelining, eso haría que incremente un poco la velocidad del proxy.

6. Conclusiones

Durante el desarrollo del trabajo se aprendieron muchas cosas sobre HTTP/1.1 y los contenidos de la materia, como por ejemplo la existencia de los headers hop by hop.

Nos ayudó a interiorizarnos con ABNF y consultar los RFC con más frecuencia.

También se logró comprender en mayor profundidad la utilidad del get y el put condicionales durante la implementación del protocolo.

Se logró comprender la complejidad que implica realizar un servidor proxy de un protocolo.

7. Ejemplos de prueba

Ejemplo con curl

Poner a correr el proxy

Abrir una terminal y ejecutar:

./httpd

```
make[1]: Leaving directory '/home/agustin/Desktop/TP_Protocolos_de_Comunicacion/http-proxy/proxy'
agustin@ubuntu:~/Desktop/TP_Protocolos_de_Comunicacion/http-proxy$ ./httpd
HTTP Proxy: Listening on TCP interface = 0.0.0.0 port = 8080
HTTP Proxy: Listening on TCP interface = :: port = 8080
Management: Listening on SCTP interface = 127.0.0.1 port 9090
Management: Listening on SCTP interface = ::1 port 9090
```

Configurar curl para que use nuestro proxy

Abrir otra terminal y ejecutar:

export http_proxy=”localhost:8080”

curl www.google.com.ar

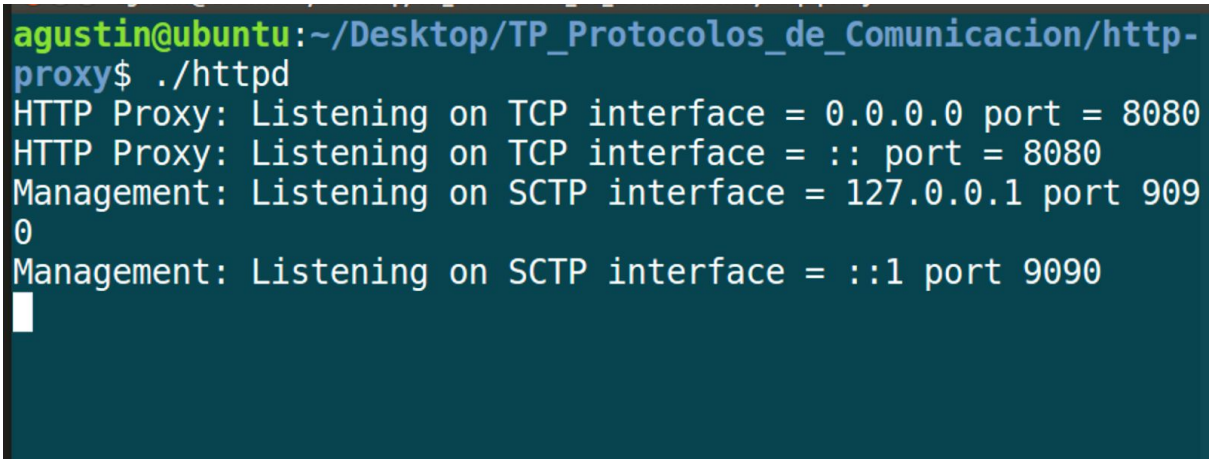
```
agustin@ubuntu:~/Desktop/TP_Protocolos_de_Comunicacion/http-proxy$ export http_proxy="localhost:8080"
agustin@ubuntu:~/Desktop/TP_Protocolos_de_Comunicacion/http-proxy$ curl www.google.com
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="es-419"><head><meta content="text/html; charset=UTF-8" http-equiv="Content-Type"><meta content="/logos/doodles/2019/celebrating-falafel-5631472089169920-law.gif" itemprop="image"><meta content="D a internacional del falafel " property="twitter:title"><meta content="D a internacional del falafel #GoogleDoodle" property="twitter:description"><meta content="D a internacional del falafel #GoogleDoodle" property="og:description"><meta content="summary_large_image" property="twitter:card"><meta content="@GoogleDoodles" property="twitter:site"><meta content="https://www.google.com/logos/doodles/2019/celebrating-falafel-5631472089169920-2xa.gif" property="twitter:image"><meta content="https://www.google.com/logos/doodles/2019/celebrating-falafel-5631472089169920-2xa.gif" property="og:image"><meta content="958" property="og:image:width"><meta content="400" property="og:image:heigh
```

Ejemplo con nc y host en request line

Poner a correr el proxy

Abrir una terminal y ejecutar:

./httpd

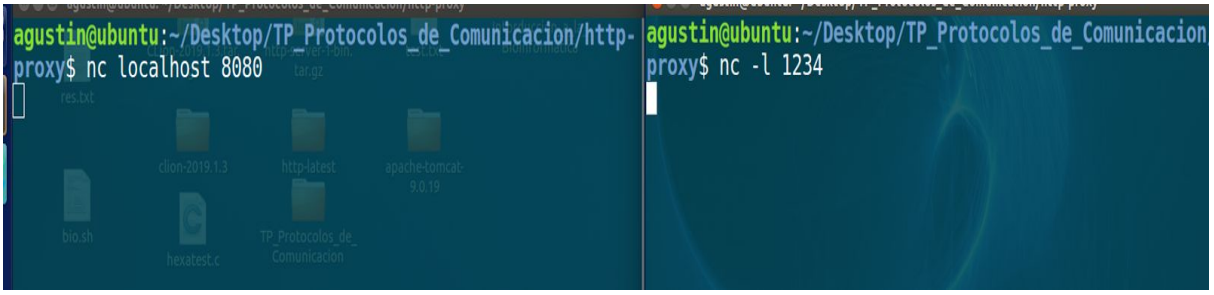


Abrir una terminal(terminal A) y escuchar un puerto que actuará como origin:

nc -l 1234

Abrir otra terminal(terminal B) y conectarse al proxy en el puerto 8080

nc localhost 8080



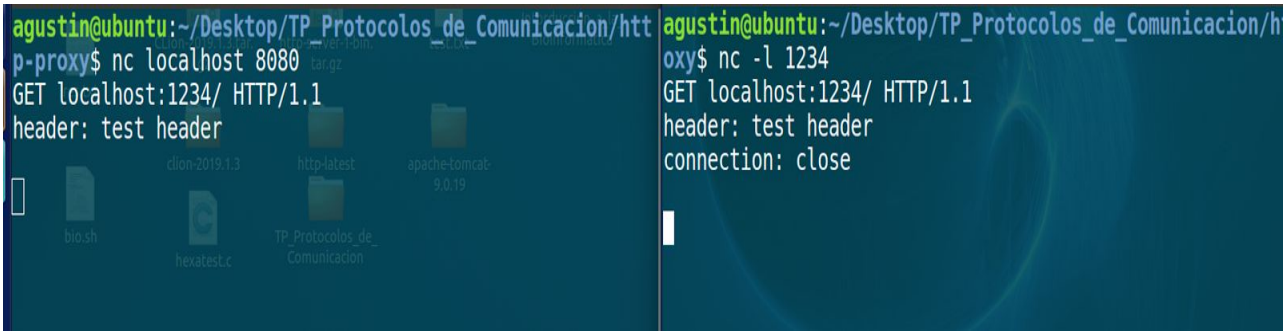
Conectarse al origin

Escribir en la terminal B:

GET localhost:1234/ HTTP/1.1LF

header: test headerLF

LF



Responder del Origin

Escribir en la terminal A:

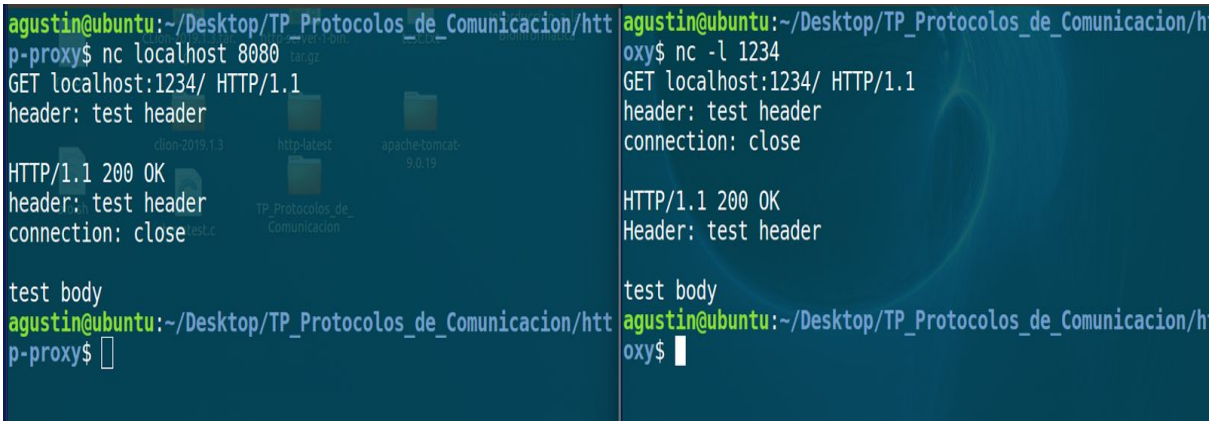
HTTP/1.1 200 OKLF

Header: last headerLF

LF

test bodyLF

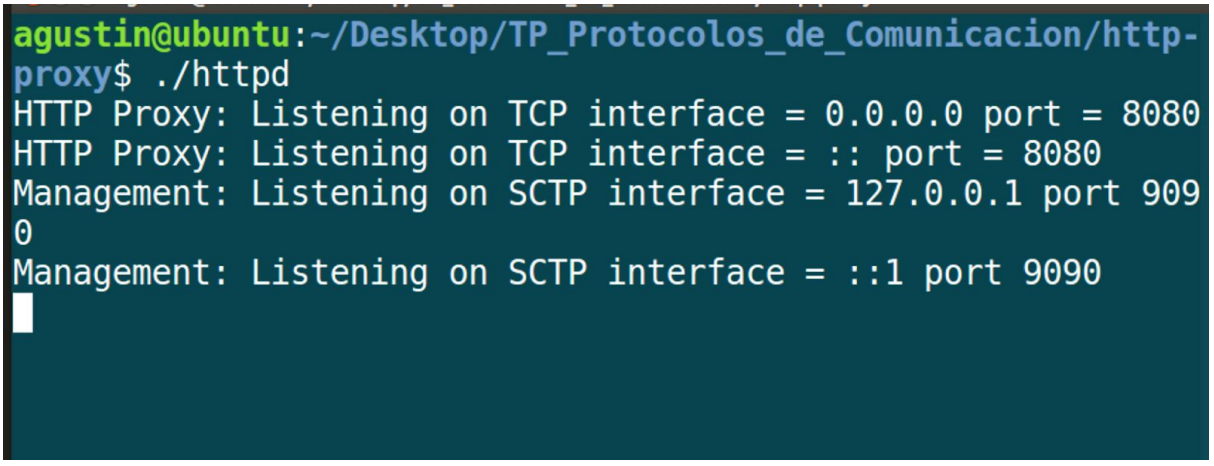
cerrar entrada (control + D)



Ejemplo con nc -C y host en header

Poner a correr el proxy

Ejecutar ./httpd

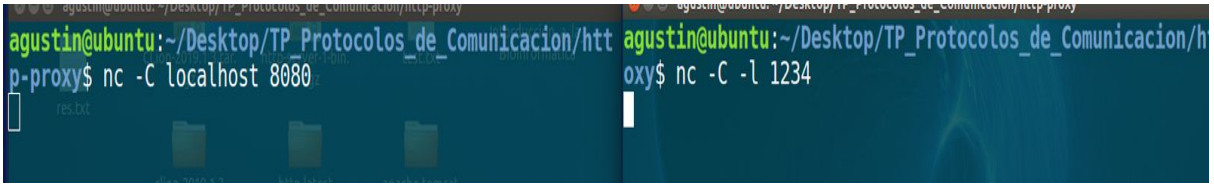


Abrir una terminal(terminal A) y escuchar un puerto que actuará como origin:

nc -C -l 1234

Abrir otra terminal(terminal B) y conectarse al proxy en el puerto 8080

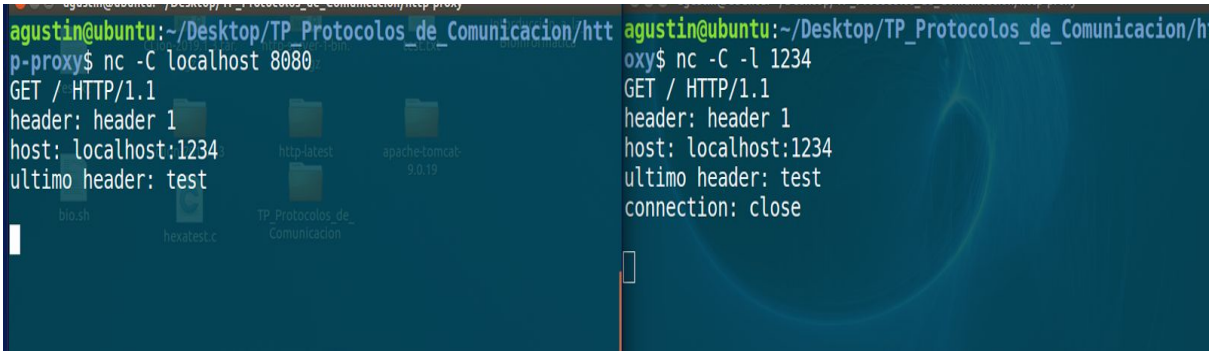
nc -C localhost 8080



Conectarse al origin

Escribir en la terminal B:

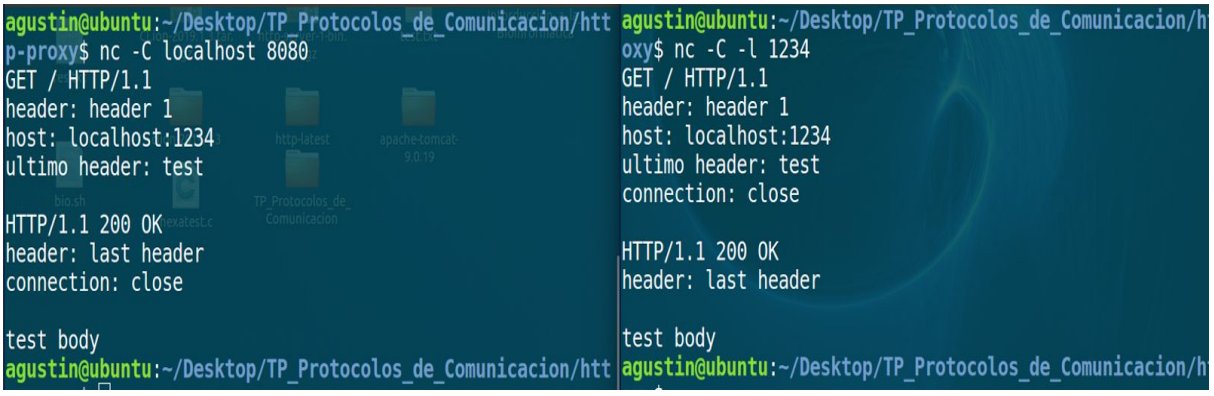
GET / HTTP/1.1CRLF
header: header 1CRLF
host: localhost:1234CRLF
ultimo header: testCRLF
CRLF



Responder del origin

Escribir en la terminal A:

HTTP/1.1 200 OKCRLF
header: last headerCRLF
CRLF
test bodyCRLF
cerrar entrada (control + D)



8. Guia de instalacion

Para instalar el programa primero deben instalarse

- make
- gcc
- libsctp-dev

sudo apt-get install make gcc libsctp-dev

Para correr el proxy pararse dentro del directorio root, ejecutar make y se generarán dos archivos ejecutables httpd (el ejecutable correspondiente al proxy) y httpdctl(el ejecutable correspondiente al manager).
Al correr httpd se generará una carpeta de logs en el mismo directorio donde se corrió el proxy que contendrá archivos de log(access.log, error.log y debug.log) que se irán generando a medida se loguea en cada uno de esos niveles.

En el readme dentro del root directory explica como instalar y ejecutar el proxy y el admin con mayor detalle.

9. Instrucciones para la configuración

Ejecución

./httpdctl [ip port]

Ip y puerto son opcionales, debe respetarse el orden y deben agregarse ambos o ninguno.

Autenticación

Usuario: manager
Contraseña: pdc69

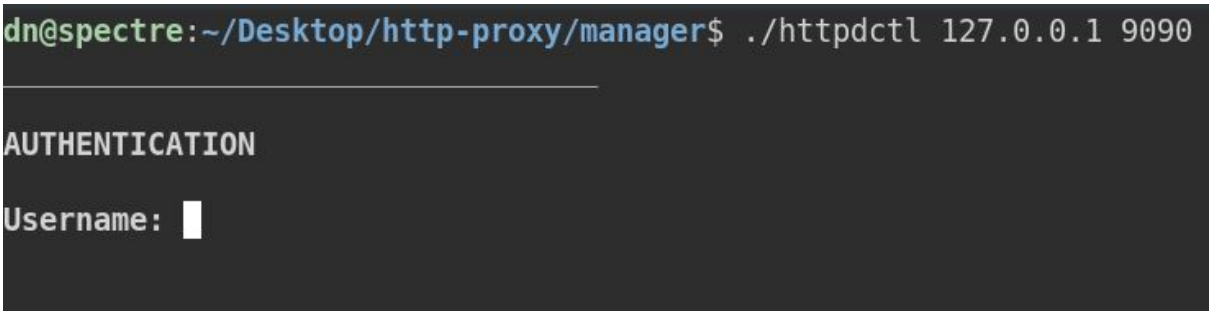
Comandos

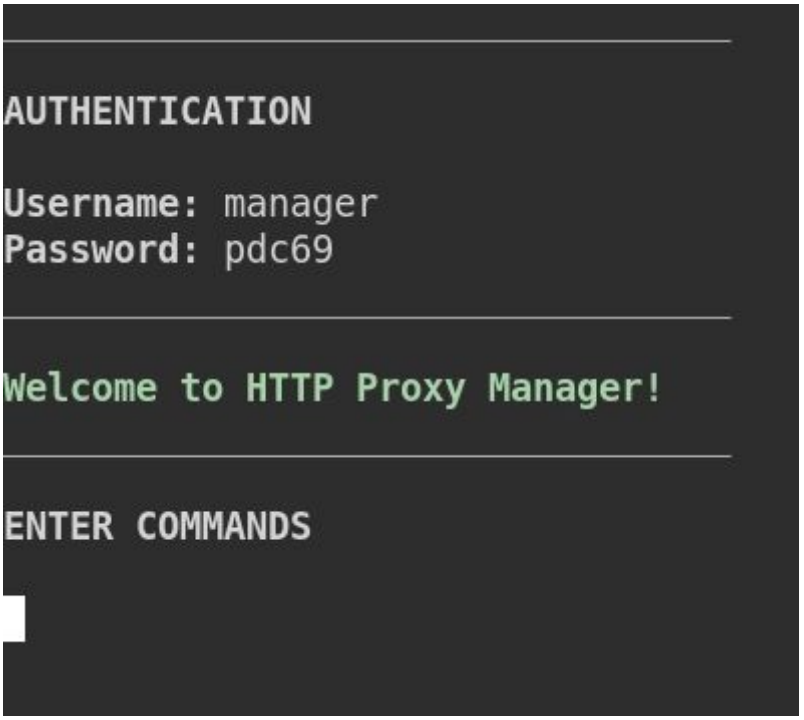
Obtiene el comando transformador
get cmd
Obtiene el media range
get mime
Obtiene el estado de las transformaciones (activado o desactivado)
get tf
Obtiene la cantidad de conexiones concurrentes
get mtr cn
Obtiene la cantidad de conexiones históricas
get mtr hs
Obtiene la cantidad de bytes transferido por el proxy
get mtr bt
Cambia el comando transformador
set cmd *command*
Agrega *media-range* a la lista de media-ranges del servidor
set mime *media-range*

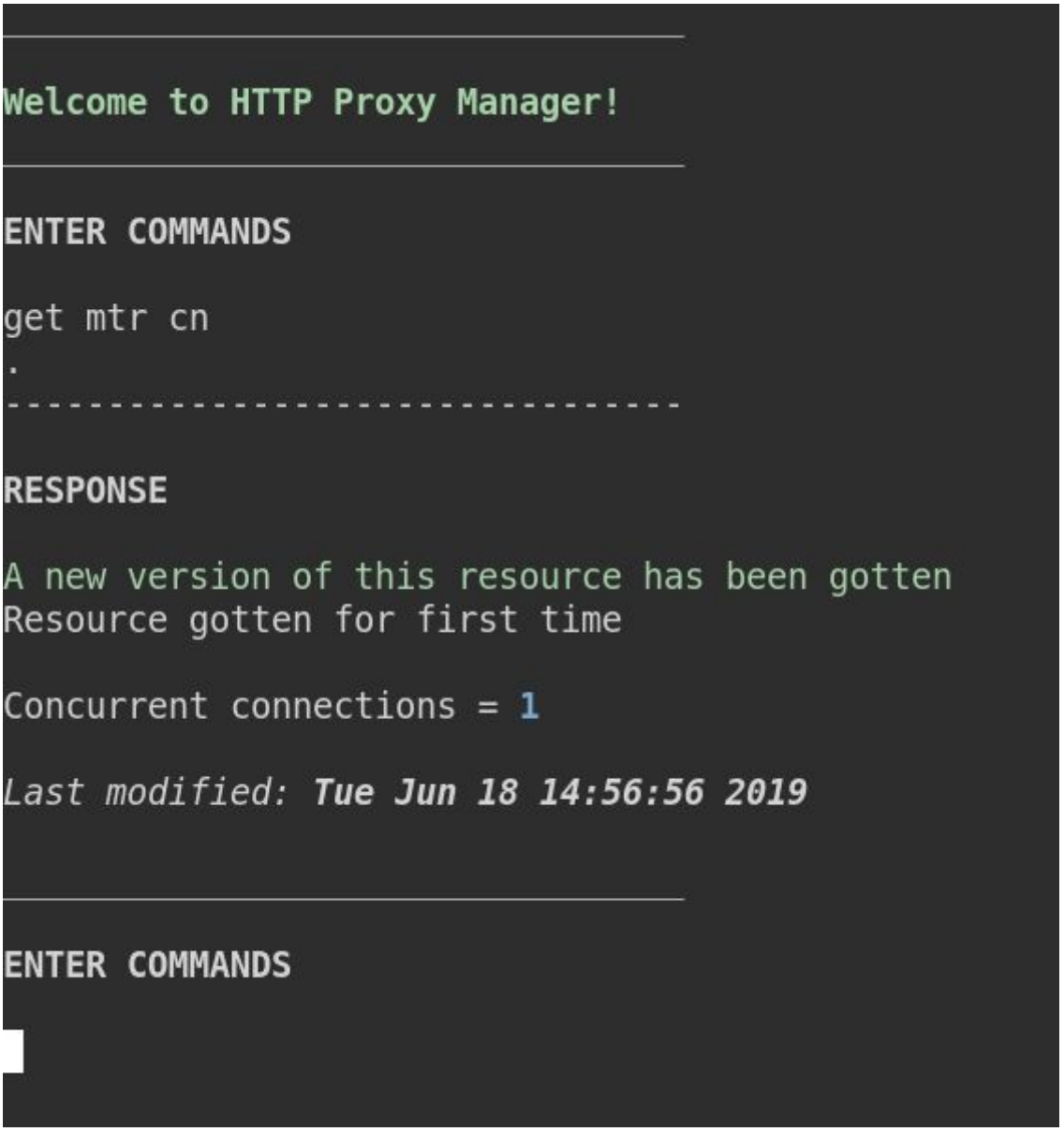
Resetea dejando vacía la lista de media-ranges del servidor
set mime
Activa o desactiva las transformaciones dependiendo si se usa el on o el off
set tf on/off
Envía un request de BYE al servidor
bye

El administrador soporta pipelining, luego de cada comando se espera **'\n'** y para terminar la lista de comandos a procesar, se manda **'.'** seguido de **'\n'**.

10. Ejemplos de configuración y monitoreo






Welcome to HTTP Proxy Manager!
ENTER COMMANDS
get mtr cn
.
RESPONSE
A new version of this resource has been gotten
Resource gotten for first time
Concurrent connections = 1
Last modified: Tue Jun 18 14:56:56 2019
ENTER COMMANDS

```
RESPONSE

A new version of this resource has been gotten
Resource gotten for first time

Concurrent connections = 1

Last modified: Tue Jun 18 14:56:56 2019


_____________________________________

ENTER COMMANDS

get mtr cn
.
---------------------------------

RESPONSE

You already had the last version of this resource

Concurrent connections = 1

Last modified: Tue Jun 18 14:56:56 2019


_____________________________________

ENTER COMMANDS

```

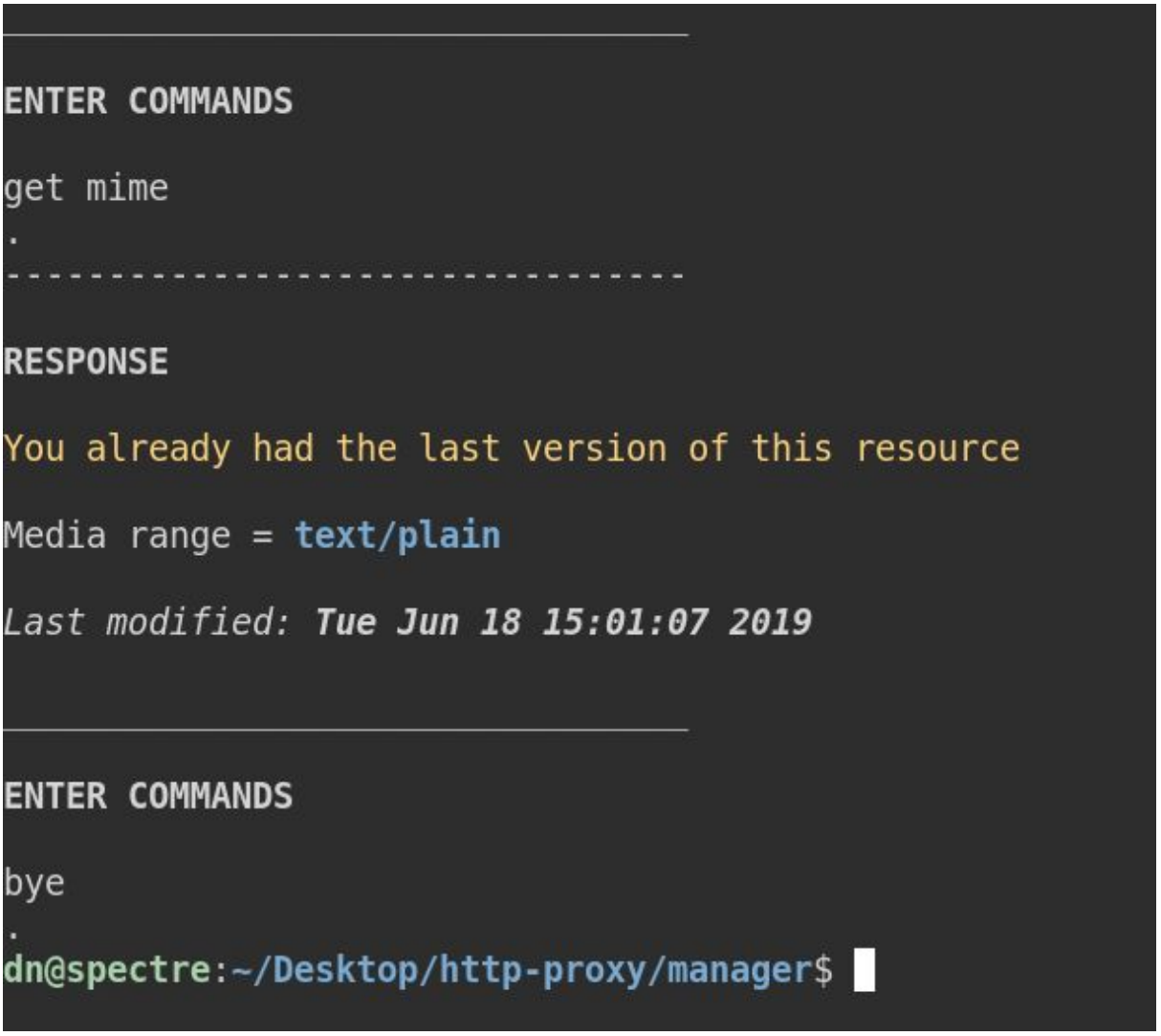
```
_____________________________________

ENTER COMMANDS

get mtr cn
get mtr hs
get mtr bt
.
---------------------------------

RESPONSE

You already had the last version of this resource

Concurrent connections = 1

Last modified: Tue Jun 18 14:56:56 2019

---------------------------------

RESPONSE

A new version of this resource has been gotten
Resource gotten for first time

Historic connections = 1

Last modified: Tue Jun 18 14:56:56 2019

---------------------------------

RESPONSE

A new version of this resource has been gotten
Resource gotten for first time

Bytes transfered = 0

Last modified: Tue Jun 18 14:56:35 2019

_____________________________________
```

```
ENTER COMMANDS

set mime text/plain
.
--------------------------------

RESPONSE

Your resource is not up to date
You need to get the last version of the resource to be allowed to modify it
```

Se puede ver lo mencionado anteriormente, un GET y un SET en pipelining, al ser del mismo estado del cliente, el SET no es exitoso

```
ENTER COMMANDS

get mime
set mime text/plain
.
--------------------------------

RESPONSE

A new version of this resource has been gotten
Resource gotten for first time

Media range =

Last modified: Tue Jun 18 14:56:35 2019

--------------------------------

RESPONSE

Your resource is not up to date
You need to get the last version of the resource to be allowed to modify it
```

```
_____________________________________

ENTER COMMANDS

set mime text/plain
.
---------------------------------

RESPONSE

Operation successfully performed. You overrided the resource!


_____________________________________

ENTER COMMANDS

get mime
.
---------------------------------

RESPONSE

You already had the last version of this resource

Media range = text/plain

Last modified: Tue Jun 18 15:01:07 2019


_____________________________________
```



11. Documento de diseño del proyecto

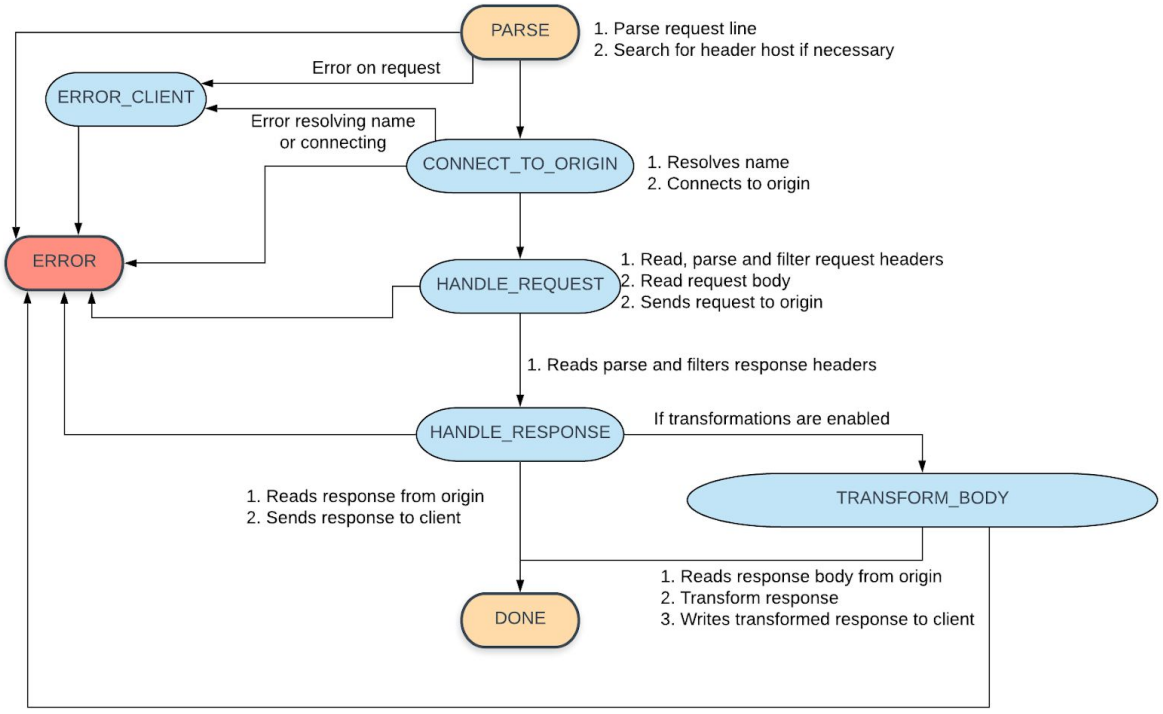


Figura 1 Estados del proxy

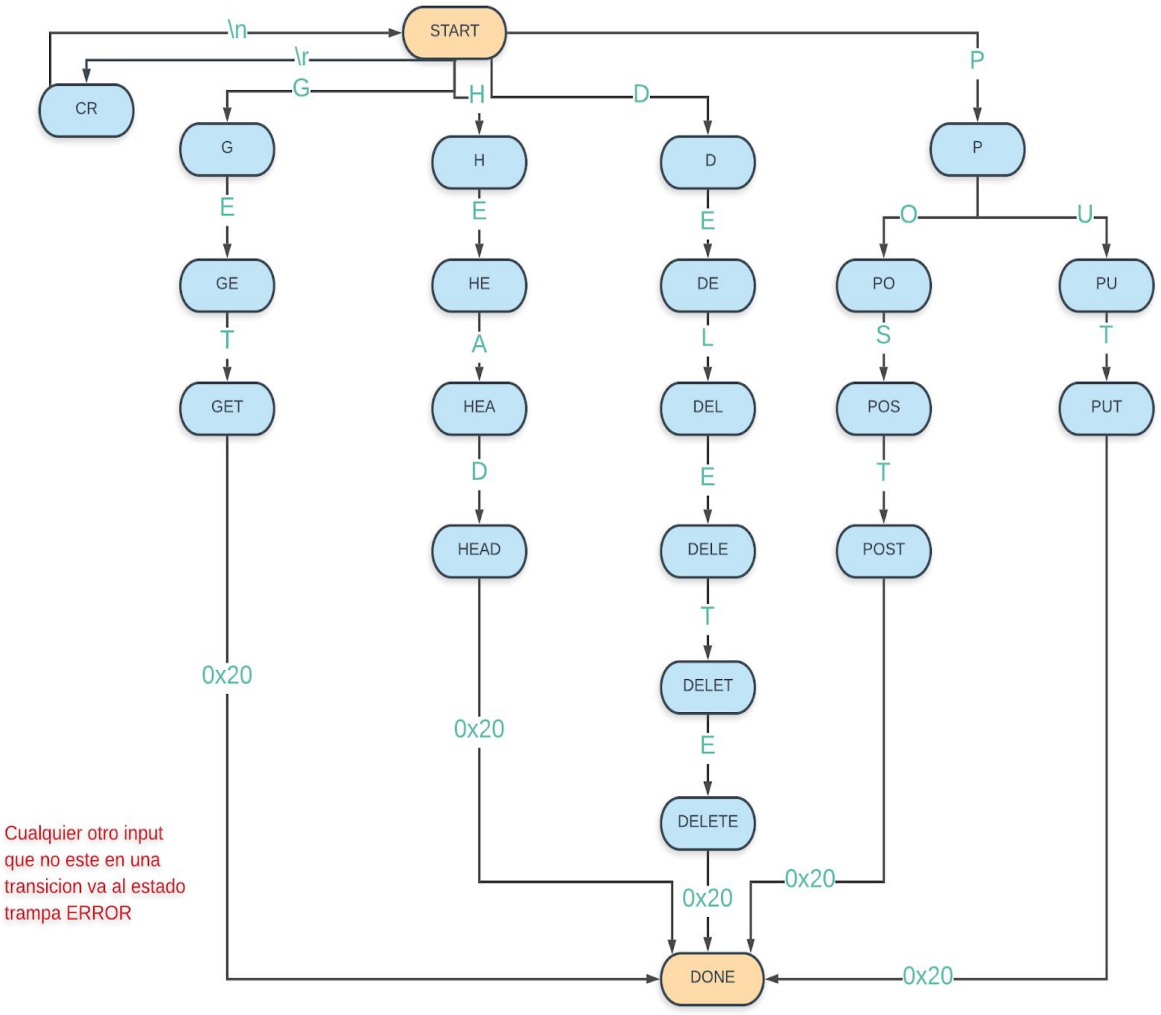


Figura 2. Method Parser

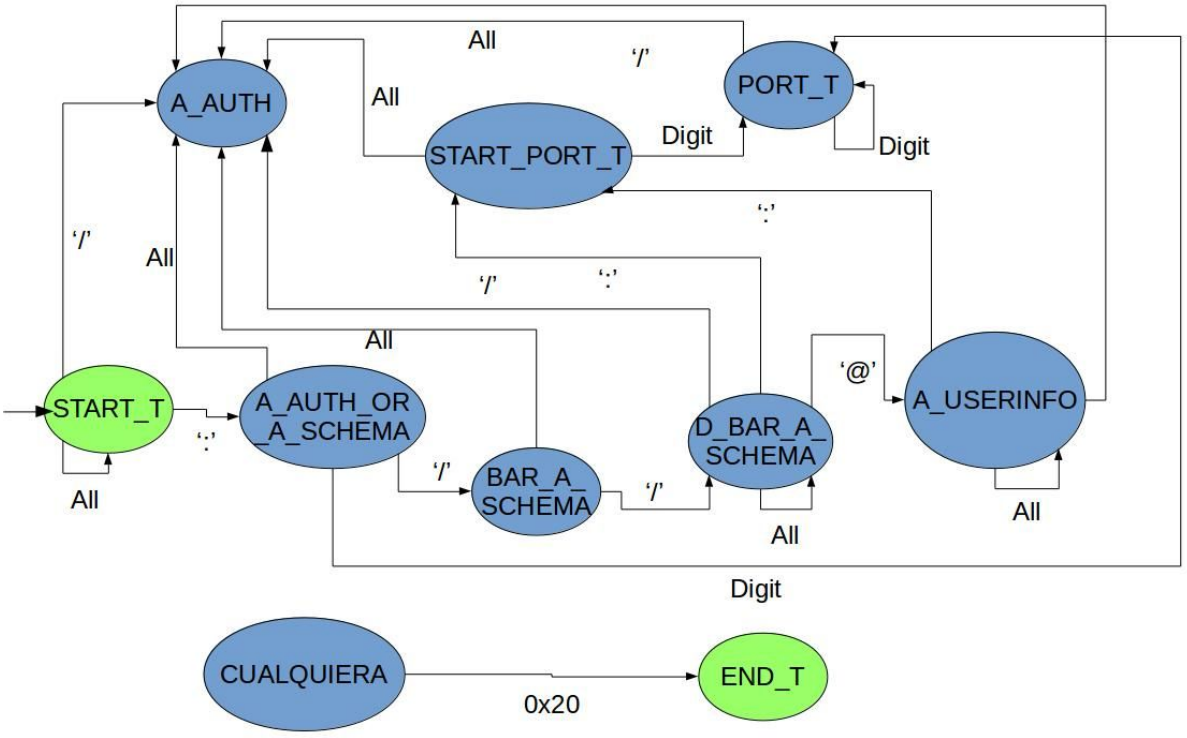


Figura 3. Target Parser

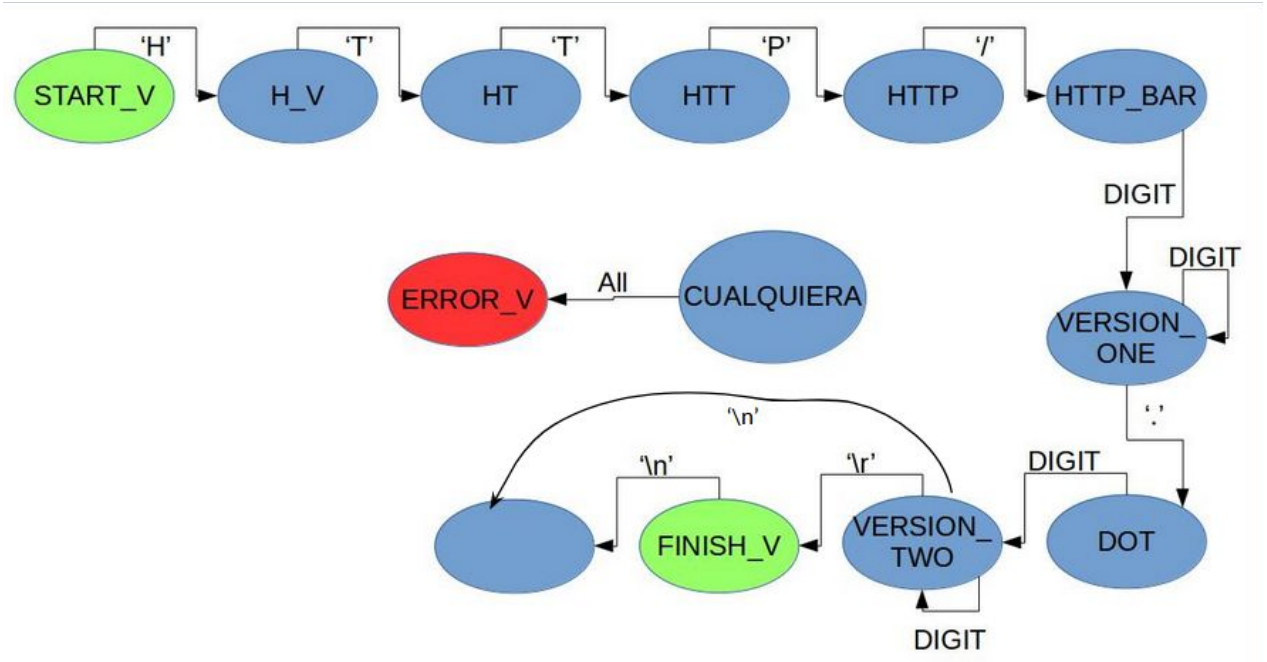


Figura 4. Version Parser

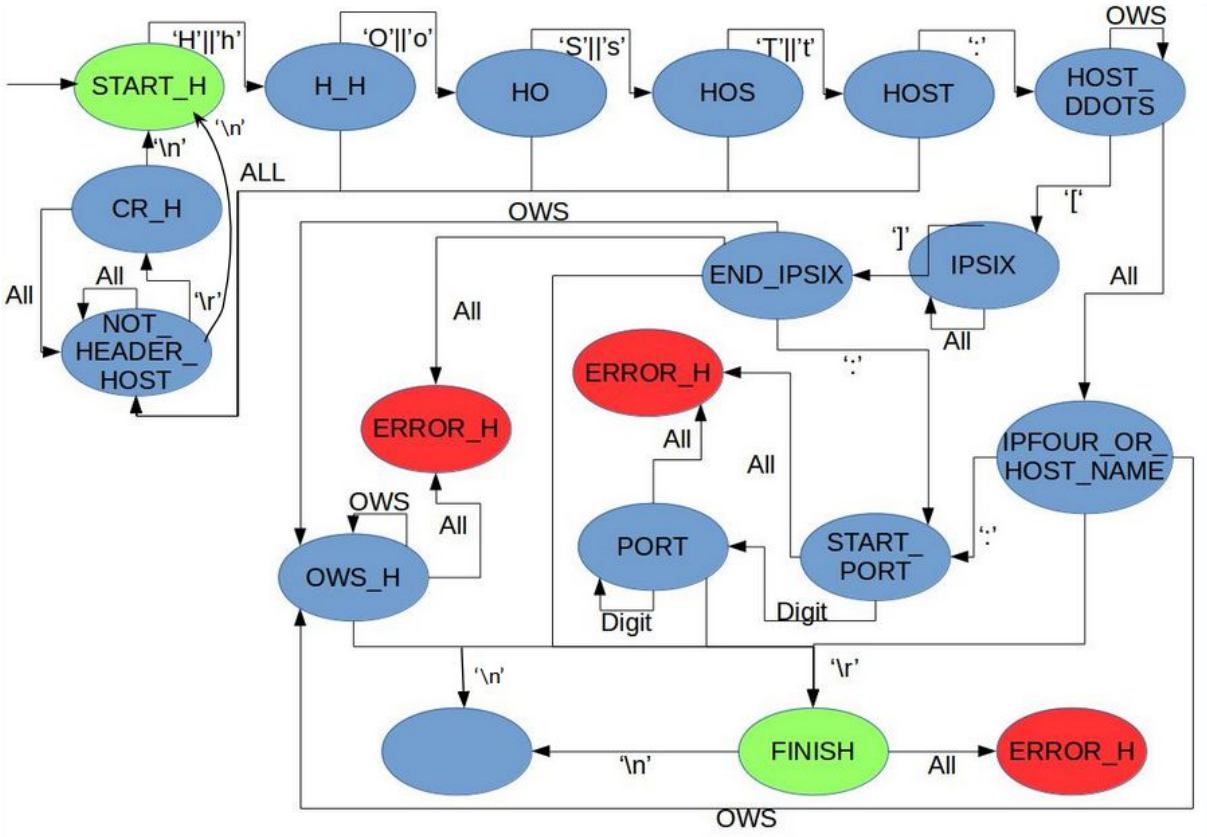


Figura 5. Header Parser

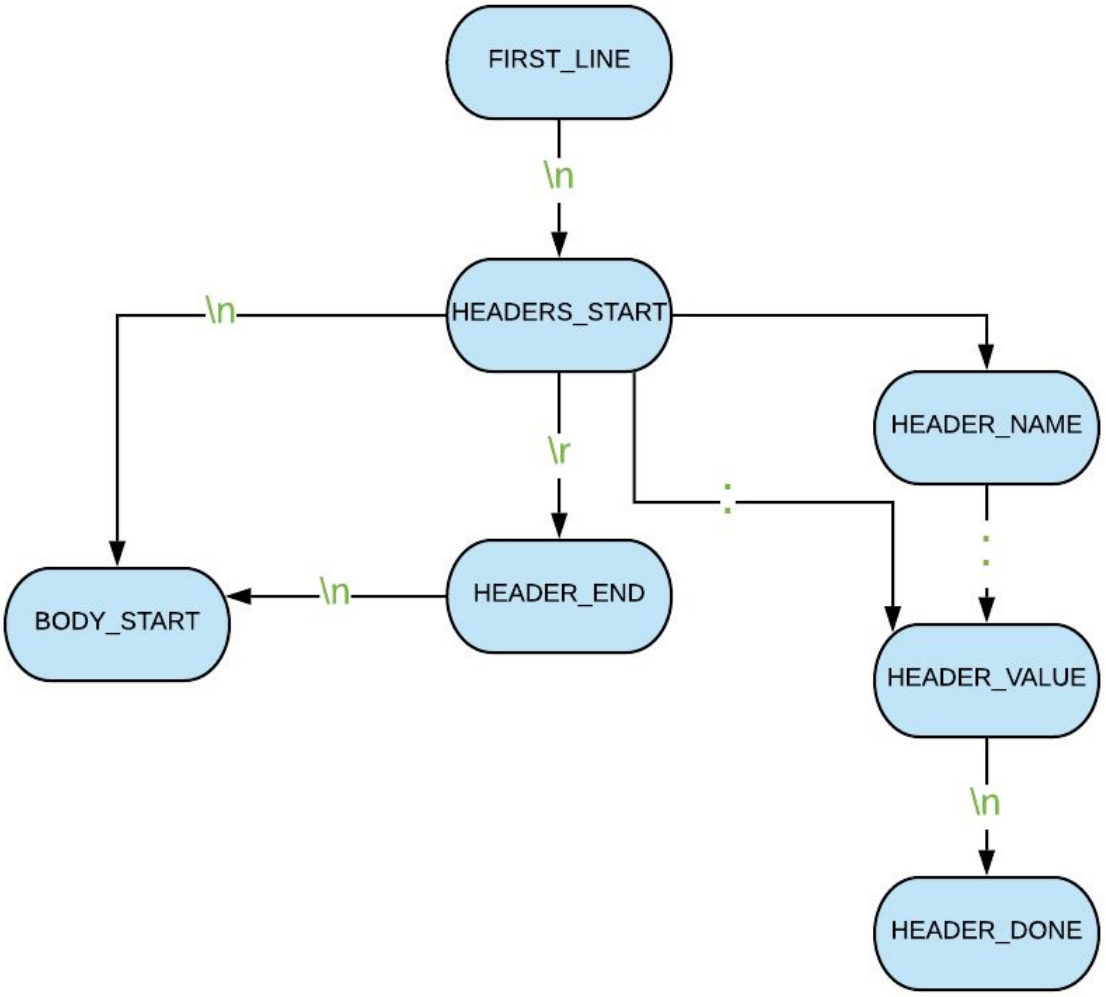


Figura 6. HeadersParser máquina de estados

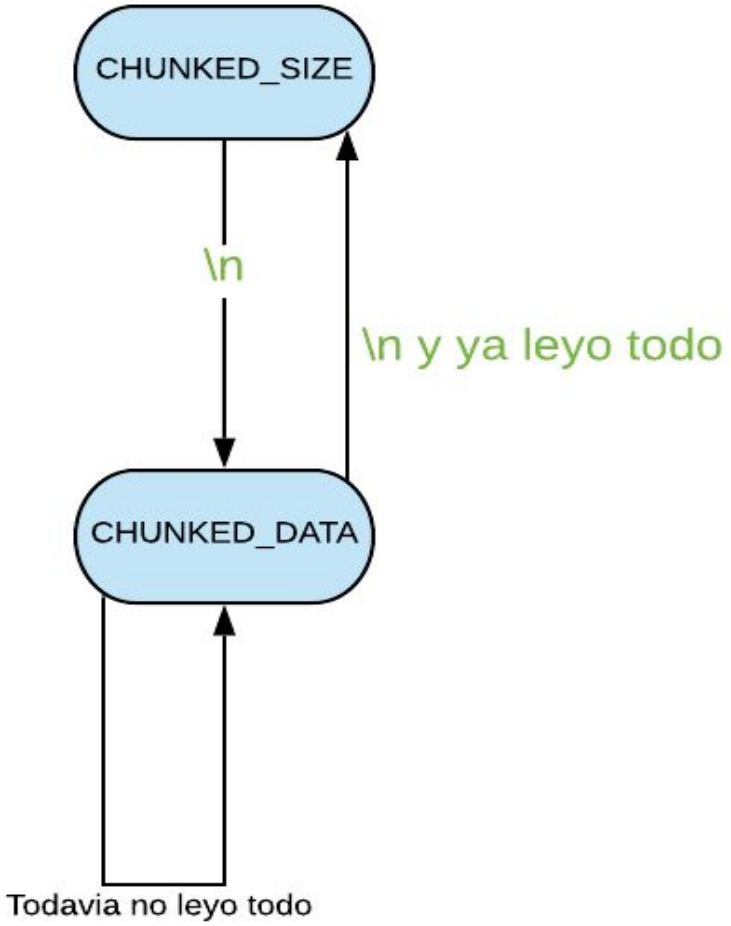


Figura 7. UnchunkedParser state machine