Trabajo Práctico Especial

Proxy HTTP

Protocolos de Comunicación

Barruffaldi, Carla - 55421
Bianchi, Luciano - 56398
Cerdá, Tomás - 56281
Perazzo, Matías - 55024

Introducción

El objetivo del trabajo fue implementar un proxy HTTP, usando Java NIO, el cual que pueda ser usado desde distintos exploradores. Se tuvo en cuenta que se puede usar el proxy de dos formas distintas: Se le puede indicar al User Agent que realice sus pedidos a través del proxy HTTP, especificando IP y puerto del mismo. En tal caso, el User Agent es consciente de que se está comunicando con un proxy. Dadas estas condiciones, según el RFC 2616, sección 5.1.2, es REQUERIDO que el User Agent envíe la URL absoluta correspondiente a la request. El proxy, entonces, obtiene así la URL que debe servir.

Otra posibilidad es que el proxy se comporte de forma transparente: el User Agent no es consciente que se está comunicando con un proxy. Por ejemplo, puede ser un reverse-proxy. En este caso, para obtener la URL que debe servir, debe obtener el host a través del header Host en el caso de que no se envíe la URL absoluta como suele ser el caso. Con el host obtenido, la URL a servir sería la concatenación del host con la URL relativa.

Diseño

La aplicación funciona haciendo uso de Java NIO, seleccionando continuamente, en un único thread, las keys que están listas para ser atendidas. Dichas keys llevan como attachment un objeto Handler, que es el encargado de realizar una acción sobre dichas keys, dependiendo del motivo para el cual fueron seleccionadas.

Existen tres tipos de Handlers:

- `ClientHandler`[1] - maneja la conexión entre el cliente y el proxy.
- `ServerHandler` - maneja la conexión entre el proxy y el servidor al que se hace la request.
- `ProtocolHandler` - maneja la conexión entre un cliente del protocolo. A través de este handler se interpretan todas las funcionalidades del protocolo.

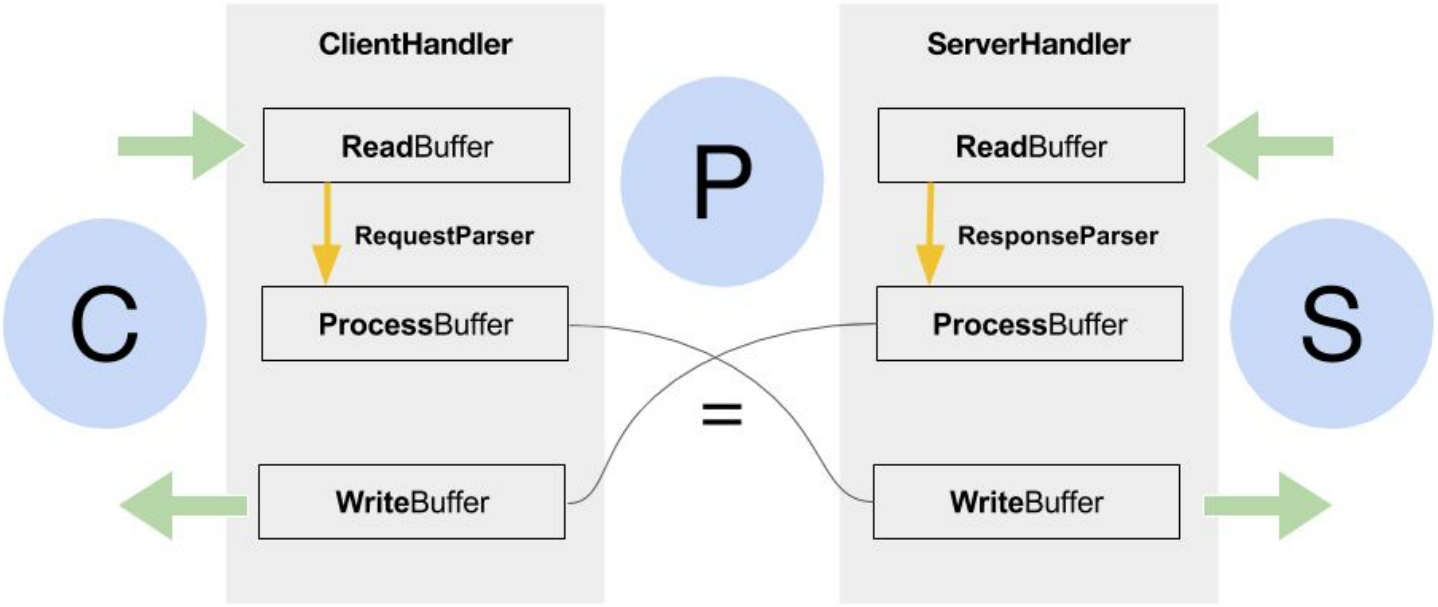
Los `ClientHandler` y `ServerHandler` están relacionados y conectados entre sí. El `ClientHandler` lee la request del cliente, la guarda en un buffer (*readBuffer*) y la parsea usando `RequestParser.` Dicho parser, a su vez, deja en otro buffer (*processedBuffer*) la request parseada. En ciertas situaciones, que se detallarán más adelante, la request que se envía al servidor no es exactamente igual a la que envió el cliente. Es por esto que la request, procesada y modificada, se deja en un buffer distinto al *readBuffer*.

El `ServerHandler` tiene un *writeBuffer*, con la misma referencia que el *processedBuffer* del `ClientHandler`. Cuando el handler del servidor puede escribir, lo hace directamente desde éste *writeBuffer*, que ya fue procesado en el handler del cliente.

[1] A lo largo del informe, para mayor claridad, se referirá como ClientHandler a la clase que en el código fuente se llama *HttpClientProxyHandler*. Lo mismo se hará con ServerHandler, en realidad llamada *HttpServerProxyHandler*.

Al leer una respuesta, la guarda en un r*eadBuffer*, y hace algo análogo a lo que se hace en el request, sólo que esta vez usando el `ResponseParser`.

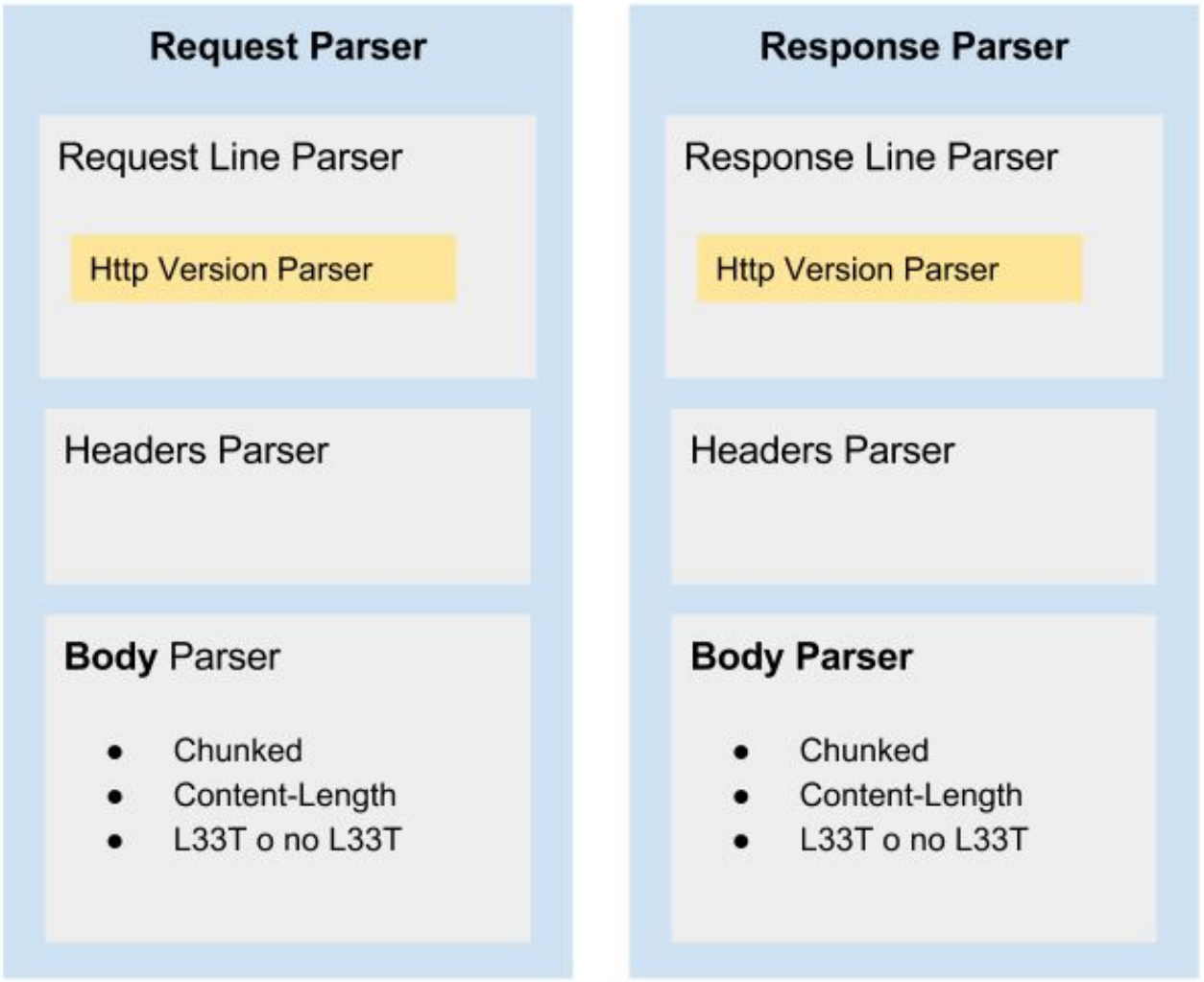
Se puede notar que este uso compartido de buffers puede llevarse a cabo de forma efectiva gracias a que se utiliza un único thread, evitando así cualquier tipo de problema que pueda surgir con la concurrencia.



Parsers

Los parsers son implementados como una máquina de estados tradicional. Leen byte a byte desde un buffer dado, cambiando su estado interno acordemente. Los parsers fueron implementados de manera tal que funcionan de igual manera sin importar la cantidad de bytes que deben procesar en cada pasada. Esto permite que las requests y responses se vayan procesando a medida que se leen, sin necesidad de tenerlas guardadas enteras.

Se notó que las request y las responses tienen ciertas partes en común, por lo que los parsers de request y response se implementaron de manera tal que están compuestos por otros parsers más pequeños. De ésta manera se evitó repetir código y se logró una mayor modularización.

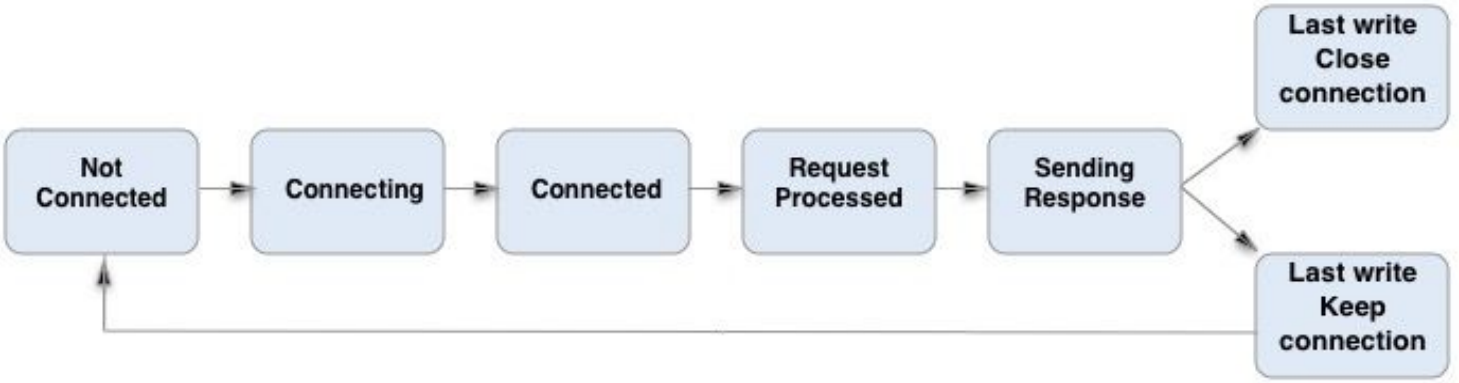


Cabe destacar que los parsers dan la posibilidad de remover y agregar headers durante el procesamiento, como también guardar el contenido de los headers que se indiquen. Se decidió que los parsers sean los encargados de realizar estas modificaciones porque son los únicos que conocen la estructura de los mensajes HTTP, y pueden modificar las requests y los responses eficientemente, ya que lo hacen a medida que leen byte a byte el mensaje.

Estados Handlers

Se hizo uso del State Pattern para describir los distintos estados en los que puede estar un handler. Se define para los estados un único método ***handle()*** el cual es invocado luego de cada escritura y lectura. En cuanto la lectura, el método se invoca luego de que el parser procese el contenido del *readBuffer* almacenándolo en el *processedBuffer*.

ClientHandler



ServerHandler

Conexiones Persistentes

Se implementa la funcionalidad de conexiones persistentes tanto del lado del cliente como del servidor.

Para su implementación se hace uso de un método ***reset()*** tanto en el `ClientHandler` como el `ServerHandler` que restauran dichos handlers a su estado inicial y registran su SelectionKey asociada a la operación que corresponda.

A su vez, se analizan los headers de request y response en búsqueda de **Connection: keep-alive**. Si no se encuentra dicho header acompañado de dicho valor, lógicamente no se persiste la conexión y consiguientemente se cierra el socket asociado al handler.

Cabe aclarar que no es necesario que ambos, el cliente y el servidor responsable de satisfacer la request del mismo, incluyan **Connection: keep-alive** en sus headers. Por ejemplo, si en los headers de la request esto no se encuentra, al procesarla se agrega el header **Connection: keep-alive** para así poder mantener la persistencia con el servidor a pesar de que no se mantenga con el cliente.

En síntesis, las persistencias cliente-proxy y servidor-proxy son independientes entre sí.

Cliente

En cuanto a la implementación del lado del cliente, basta simplemente hacer uso del método ***reset()*** sobre el `ClientHandler` cuando la response del servidor haya finalizado. Este método vuelve a registrar la SelectionKey asociada para lectura, en lugar de cerrar el socket como sería en el caso ausente de persistencia.

El mayor cuidado a tener en cuenta es en el caso de que la response no venga acompañada de **Content-length** o **chunked**; imposibilitando determinar la finalización de la misma sin cerrar la conexión. Para esto se presentan dos soluciones:

- Cerrar la conexión al cliente (a pesar de la presencia de **Connection: keep-alive**);
- Alterar la respuesta del servidor, haciéndola llegar al cliente de forma **chunked**.

Se optó por la primer opción de cerrar la conexión al cliente debido a su simplicidad.

Servidor

La conexión con el servidor se guarda en un caché una vez que se termina de procesar la respuesta, no se espera a que se envíe toda la respuesta al cliente.

En cuanto a la implementación del lado del servidor, se utiliza un caché de conexiones administrado por el singleton `ConnectionManager`. Es responsable de establecer nuevas conexiones como de reutilizar las existentes y limpiar las que no se utilizan pasado un timeout.

Hace uso de un mapa de direcciones (par IP-puerto) a una cola de tamaño fijo en la cual se almacenan las conexiones establecidas para dicha dirección. Esto permite almacenar múltiples conexiones hacia un mismo servidor pero con un límite.

El método ***connect()*** abstrae el proceso de conexión o reutilización de conexión existente. Al intentar reutilizar una conexión, primero verifica que dichas conexiones sigan siendo válidas realizando un read no bloqueante sobre el socket de la conexión y verificando que la cantidad de bytes leídos haya sido 0 y no se haya recibido EOF.

Logs

A lo largo de todo el desarrollo del proyecto se hizo uso de la librería de Logback4J para loguear información de debug a salida estándar, permitiendo seguir paso a paso la ejecución del proceso. Estos logs son de uso exclusivo de desarrollo y no se imprimen en la versión de deploy del programa.

En cuanto a los logs requeridos según las consignas del trabajo práctico, se optó por loguear a disco dos tipos de logs: acceso y error en archivos separados, se forma análoga a como lo hace nginx. Estos logs se guardan en un archivo diferente según la fecha también, similar a los de Tomcat.

Formato log de acceso:
`[hh:mm:ss]{ IP cliente} - {Request} [{Host:puerto servidor}] {Status code} UA:{User Agent} SV:{Server}`

Formato log de error:
`[hh:mm:ss] {Mensaje de error}, client: {IP cliente}, request: {Request}, host: {Host del servidor}`

Donde dice Request se refiere a la primer línea de la request del cliente, no la request completa.

Puede darse el caso en los logs de error que no se presente la request o el host si el error surgió de forma previa a que dichos datos puedan ser extraídos.

Finalmente, en cuanto a la implementación, se hace uso del singleton `ProxyLogger` que se encarga de bajar a disco la información y construir el mensaje pero en un thread diferente a partir del `ExecutorService SingleThreadExecutor`.

Archivo de configuración

La aplicación dispone de una forma de determinar una serie de configuraciones iniciales mediante un archivo .properties, que permite cambiar, entre otras cosas, el puerto en el cual escucha el proxy, el tamaño de los buffers que usan los parsers y los handlers, y los tiempos de espera entre cada limpieza de las conexiones guardadas.

Protocolo POPIS

Características

- Es un protocolo inspirado en REDIS y POP3.
- Las respuestas a una request cuentan con el prefijo “-” o “+” dependiendo de si hubo un error o no en la request.
- Una request puede estar formada por varios comandos, terminando en el comando end. El proxy termina su respuesta con un end y cierra la conexión.
- Las requests y los headers deben finalizar con CRLF.
- Una request CRLF (sin contenido) no es válida.
- Case insensitive.
- Si un header recibe argumentos primero se indica la cantidad y luego los argumentos.

ABNF

Comando	=	1*((HeaderSinArg CRLF) / (HeaderConArg CRLF ArgCount CRLF Args))	
HeaderSinArg	=	“proxy_buf_size” / “l33t_enable” / “l33t_disable” / “is_l33t_enabled” / “client_bytes_read” / “client_bytes_written” / “client_connections” / “server_bytes_read” / “server_bytes_written” / “server_connections” / “metrics” / “ping” / “end”	
HeaderConArg	=	“set_proxy_buf_size” / “method_count” / “status_code_count”	
ArgCount	=	%d42 %d49-49.56	;corresponde al rango 1-18
Args	=	1*((StatusCode / HttpMethod / BufferSize) CRLF)	
StatusCode	=	%d51.48.48-53.57.57	;corresponde al rango 300-599
HttpMethod	=	“GET” / “POST” / “HEAD” / “OPTIONS” / “PUT” / “DELETE” / “TRACE”	
BufferSize	=	%d50.53.54-49.48.52.56.53.55.54	;corresponde al rango 256-1024x1024

Headers

Métricas

Corresponden a los headers *client_bytes_read, client_bytes_written, client_connections, server_bytes_red, server_bytes_written, server_connections, method_count* y *status_code_count*. Los mismos devuelven información sobre métricas del proxy. También se cuenta con el header *metrics* el cual devuelve información sobre todas las métricas ya mencionadas

Setters

Son aquellos que configuran propiedades del proxy o flags del mismo. Estos son *set_proxy_buf_size, l33t_enable y l33t_disable*.

Getters

Proveen información sobre las propiedades del proxy o flags del mismo. Estos son *proxy_buf_size y is_l33t_enabled*.

Debugging

Son utilizados para verificar que las funcionalidades básicas estén funcionando correctamente. El único header de debug es *ping* y comprueba que el servidor pueda recibir una request y generar una response.

Finalización de request

Indica el final de la request. El header que lo hace es *end*.

Errores

Tipos

El protocolo cuenta con los siguientes errores:

- `NOT_VALID`: ocurre cuando el mensaje no satisface la sintaxis descrita en el protocolo.
- `TOO_LONG`: se genera cuando el mensaje es muy largo.
- `NO_MATCH`: ocurre cuando el mensaje tiene sintaxis correcta y el largo es adecuado, pero el método ingresado no corresponde con uno definido por el protocolo.

Clasificación

El protocolo define la siguiente clasificación de errores:

- *Recuperables*: ocurre cuando se genera un error que puede ser restablecido. La aplicación puede descartar la línea con el error, e interpretar las siguientes.
- *No recuperables*: son aquellos errores que finalizan la request.

El mismo no define qué errores pertenecen a cada clasificación, sino que esto debe ser determinado por la implementación.

Respuesta

Sin error en la request

El nombre de cada método de la request va a ser precedido con un '+'. En el caso de que se necesite colocar el valor de una métrica o de un flag se pone a continuación colocando ':', un espacio y el valor.
Por ejemplo: *client_bytes_written*\r\n generaría el output *+client_bytes_written: 100*\r\n

Con error en la request

Se recibirá un mensaje precedido por un '-', indicando el tipo de error generado entre corchetes '[tipo_de_error]'. En este caso no se va a colocar ningún tipo de valor a continuación dado que no se recibió algo válido.
Si se trata de un error no recuperable, en la respuesta se va a indicar hasta dónde se parseó el input antes de cerrar la conexión indicándose a partir de “[...]”.
En el caso de los errores recuperables, se descarta la línea y el tipo de respuesta a la línea siguiente es independiente de la anterior.
Por ejemplo: *client_bytes_dropped*\r\n generaría la respuesta:
-[NO_MATCH]client_bytes_d[...]\r\n
+end\r\n

Ejemplo de request-response sin error

Request

client_bytes_read\r\n

server_bytes_written\r\n

method_count\r\n*3*\r\n

GET\r\n*POST*\r\n*HEAD*\r\n

status_code_count\r\n*2*\r\n*302*\r\n*404*\r\n

end\r\n

Response

+client_bytes_read: 10099\r\n

+server_bytes_written: 9999\r\n

+method_count\r\n*+*3*\r\n

+GET: 30\r\n+POST: 10\r\n+HEAD: 5\r\n

*+status_code_count\r\n+*2\r\n+302: 10\r\n+404: 7\r\n*

+end\r\n

Ejemplo de request-response con error

Request

client_bytes_read\r\n

server_bytes_written\r\n

*method_count!!\r\n*3\r\n*

GET\r\nPOST\r\nHEAD\r\n

end\r\n

Response

+client_bytes_read: 10099\r\n

+server_bytes_written: 9999\r\n

-[NOT_VALID]method_count![...]\r\n

+end\r\n

Ventajas

- Fácil de implementar.
- Comprensible para quien lo utiliza, no solo por ser un protocolo orientado a texto sino que además cuenta con una sintaxis clara y fácil de usar.
- Fácil detección de posibles errores en el request. En especial errores de sintaxis cuando una persona usa directamente el protocolo mediante netcat, por ejemplo.
- En el caso de que se genere un error no recuperable, se indica hasta dónde se parseo el mensaje, permitiendo identificar y reemplazar el comando que generó el error.

Posible extensiones

- Contar con errores más descriptivos.
- Agregar headers que aporten más funcionalidad al protocolo, como headers de blacklist, de promedios de conexiones o bytes transferidos por unidad de tiempo.
- Contar con un método de autenticación para su uso.

Desafíos durante el diseño y la implementación

Manejo de buffers

Manejo adecuado la registración y desregistración de SelectionKey al llenarse o vaciarse un buffer, como también las operaciones de ***flip()*** y ***clear()/compact()***. Un mal manejo de estas operaciones pueden llevar al estado inconsistente del proxy.

Es por esto que para el caso de manejo de claves se realiza en clases de poco código como son las clases de estado de los handlers y en los métodos que settean el estado de un handler.

Para el manejo de flip, clear y compact, estas operaciones se realizan estricta y únicamente en la clase abstracta `HttpHandler`, también compuesta de pocas y simples línea de código, librando a las implementaciones concretas del uso de dichas operaciones.

Claridad de código

Debido a los problemas que surgen a medida del desarrollo del proyecto, arreglarlos manteniendo un nivel adecuado de claridad en el código resultó dificultoso.

Debido a esto, el proceso de desarrollo tomó la siguiente forma: los problemas se fueron resolviendo adquiriendo así un conocimiento completo de sus características y resolución. Una vez que el problema se encontraba resuelto, se miró al código como un todo, buscando reutilizar y reorganizar el código mediante las abstracciones adecuadas con el objetivo de mejorar su claridad. Finalmente, se llevaba a cabo el refactor, manteniendo la funcionalidad intacta.

Ejemplos concretos de esto se puede apreciar en el desarrollo de los parsers y los estados de los handlers.

Conexiones persistentes

Uno de los problemas presentados al implementar la funcionalidad de conexiones persistentes fue el manejo adecuado de conexiones obsoletas y que no queden las keys en el Selector estando obsoletas.

A su vez, al usar una queue de tamaño fijo donde al agregar una nueva conexión con la queue llena simplemente se pisa la referencia a la conexión más vieja, esta conexión nunca se cerraba, consumiendo sockets y lugares en el Selector. Eventualmente surgía una excepción de too many open files, cuya causa fue difícil de encontrar.

Fiabilidad en el testing

No es trivial hacer tests de carga y de performance en una aplicación como ésta, debido a la volatilidad en el comportamiento de varios de los factores involucrados (calidad de conexión, velocidad del cliente y servidor, velocidad y recursos disponibles en la computadora que corre el proxy).

Por ejemplo, los tests de carga usando JMeter -teniendo una computadora corriendo JMeter y otra en la misma red que ella corriendo el proxy y un servidor nginx, para disminuir el efecto de la varianza en la calidad de conexión- no eran demasiado confiables debido a que había una pérdida de performance en la computadora que simulaba a los clientes, ya que tenía que abrir una gran cantidad de threads en simultáneo y por razones obvias nunca iba a lograr el mismo comportamiento que varias computadoras distintas conectándose al proxy simultáneamente. Lo ideal hubiese sido disponer de la infraestructura necesaria para hacer este tipo de pruebas.

Limitaciones

Métodos soportados

Sólo soporta los métodos GET, POST, DELETE, PUT y HEAD.

Transformaciones

No realiza la transformación en casos de que una request de POST o PUT lleve el body comprimido. A su vez, con el flag de transformación prendido, se filtra el header **Accept-Encoding**, teniendo como consecuencia que todas las responses no vendrán comprimidas. Por esto se sugiere prender el flag únicamente si se esperan transformaciones.

Asimismo, solo se realiza la transformación dados los siguientes charsets (o la ausencia del mismo):

- UTF-8
- ISO-8859-1

- ASCII
- US-ASCII

Conexiones

Se decidió no limitar la cantidad de conexiones que el proxy soporta, ya que durante los tests no se pudo obtener un límite fiable. La capacidad del proxy está limitada por dos factores fundamentales: la cantidad de file descriptors que se pueden abrir para leer y escribir a los sockets, y la memoria disponible para el proceso del proxy. Cabe destacar que a pesar de rechazar nuevas conexiones cuando, por ejemplo, no se pueden abrir más sockets, el proxy sigue atendiendo correctamente a los demás cliente.

Tamaño dinámico de buffers

En cuanto al tamaño dinámico de los buffers del proxy que se pueden establecer a través del comando *set_proxy_buf_size,* el tamaño mínimo a establecer es de 512 bytes y el máximo es de 2 Mb.

Tamaño estático de buffers

Se definen a continuación los tamaños máximos de recursos como longitud de URL o contenido de headers:

- Contenido de un header: 128
- Hostname: 256
- Primer línea del request: 2048

Posibles extensiones

Caching

Una evidente mejora que se le puede incluir al proxy, guardando en memoria o en disco (se pueden contemplar cualquiera de las dos opciones dependiendo del comportamiento buscado) los recursos que piden los clientes, una característica común en este tipo de aplicaciones.

Blacklisting

Otra característica útil en un proxy es la posibilidad de bloquear el acceso a ciertos clientes o a ciertos servidores, creando "listas negras" para cada caso. Para configurar dicha funcionalidad, se puede aprovechar la forma actual de comunicarse con el proxy mediante el protocolo diseñado, incluyendo un nuevo método y pasando por parámetros los IPs o los dominios de los hosts que se quieren prohibir.

Autenticación

En la forma actual, cualquiera en la misma red que la computadora que corre el proxy, que pueda conectarse a ella por TCP, puede obtener las métricas y cambiar las configuraciones del mismo. Ésto puede traer varios problemas si se quiere usar el proxy en una red pública (como por ejemplo, la del ITBA), y puede ser solucionado requiriendo una autenticación previa (usando usuario y contraseña) para comunicarse con el proxy. Puede haber distintos niveles de autenticación, dependiendo de si sólo se quiere usar el proxy o si también se lo quiere poder modificar.

HTTPS

Dado que la mayoría de las páginas web más visitadas requieren HTTPS, implementar esta funcionalidad ampliará significativamente los casos de uso del proxy.

Multithreading

A pesar de que ,usando Java NIO, el proceso del proxy no se encuentra bloqueado en ningún momento, se podría aprovechar una computadora con varios núcleos que corra el proxy haciendo uso de multithreading. De todas maneras, implementar dicha característica en el proxy puede resultar desafiante debido a los problemas de concurrencia y sincronización que esto puede traer, y puede que la mejora en performance termine no siendo significativa.

Monitoreo real-time con un cliente especial

Una funcionalidad interesante que puede incorporar el proxy, es la posibilidad de obtener las métricas del proxy en tiempo real, en *streaming*. Puede ser una herramienta útil para un administrador de una red con mucho tráfico, ya que el proxy podría funcionar como una suerte de monitor de red.

L33t de HTML

Con un parser de HTML apropiado, podría realizarse la transformación l33t en el contenido textual de las páginas cargadas.

Conclusiones

En retrospectiva, se logró una implementación satisfactoria de la aplicación propuesta. Como se puede ver en el historial del repositorio de git, se empezó el proyecto de forma temprana, y a medida que se aproximaba la fecha de entrega, el trabajo se concentró en corregir errores y en clarificar el código.

El enfoque que se le dió al desarrollo, que se detalló en la sección de desafíos, de primero resolver el problema y después abstraer y clarificar el código, resultó sumamente efectivo a la

hora de entender los problemas que se afrontaban, en especial porque era la primera vez que se trabajaba con herramientas como Java NIO.

El resultado final consideramos que es una aplicación robusta, compuesta por código relativamente claro y fácilmente extensible.

Casos de prueba

Criterios de implementación

Se presentan a continuación las postcondiciones de los casos de prueba otorgados por la cátedra cuyo resultado depende de detalles de implementación del proxy.

010 - Múltiples requests en una misma línea

Se agrega la siguiente postcondición fruto del uso de conexiones persistentes entre proxy y cliente:

- Se reusa la conexión entre el cliente de cURL y el proxy. Se puede verificar haciendo uso del modo verboso con ***-v*** y observar la presencia de las siguientes dos líneas:
    - ** Connection #0 to host **localhost** left intact*
    - ** Re-Using existing connection! (#0) with proxy **localhost***

Puede variar el contenido **localhost** si el proxy no está corriendo de forma local.

013 - L33t Chunked gziped

Se cumple únicamente la siguiente postcondición:

- Se realiza la transformación y el contenido no está comprimido. No se presenta el header **Content-Encoding** en la respuesta.

016 - El Proxy como Origin Server

Se cumple únicamente la siguiente postcondición:

- Una implementación consciente del problema detecta la recursión antes de conectarse y respondería una respuesta dentro de la familia de los 400/500.

Pruebas de Rendimiento

Las pruebas fueron realizadas a partir del programa Apache Jmeter, versión 3.1 en una MacBook Air modelo A1466, procesador Intel i5 1.3 GHz, 4GB de memoria 1600 MHz DDR3. Mediante la configuración de JMeter, se le asignaron 2GB de heap de Java para el funcionamiento de dicho programa.

Transferencia vs Tamaño de buffer

Para cada tamaño de buffer se realizó una prueba de JMeter con un sampler HTTP realizando la petición de una iso a un servidor nginx 10.1.3 corriendo de forma local. La iso corresponde a http://releases.ubuntu.com/16.04.2/ubuntu-16.04.2-desktop-amd64.iso?_ga=2.93647517.1614020202.1497326635-225387047.1497326635, cuyo tamaño es de 1.55 Gb.

A su vez, JMeter estaba configurado para realizar toda petición al proxy.

- Número de threads: 1
- Período de subida: 1 segundo
- Contador del bucle: 5

Tamaño buffer (bytes)	Tiempo total transcurrido (mm:ss)	Kb/sec
1024	03:44	34017,23
2048	01:46	71485,24
4096	00:55	136814.04
8192	00:31	241335,71
16384	00:21	358825,54
32768	00:15	490930,35
65536	00:12	599147,43

Tabla 1.

Resulta interesante observar como los tiempos de transferencias disminuyen a la mitad y las velocidades se duplican hasta que el tamaño del buffer es de 8192 bytes. A partir de allí, si bien las ganancias son claras, no son tan predominantes como en los otros casos.

Prueba de stress

Las pruebas fueron realizadas conectando directamente mediante un cable Ethernet dos MacBook Airs idénticas, con las mismas características ya descritas. Se utilizó el mismo servidor nginx 10.1.3.

Con el objetivo de ver la degradación del tiempo de respuesta y de la latencia del proxy a medida que aumentaban los clientes concurrentes al mismo. Desde una computadora se corrió un servidor nginx y el proxy, y desde la otra se corrió JMeter.
En JMeter se generaron 100 threads, con un ramp-up time de 100 segundos, que realizaron 75 iteraciones pidiendo un recurso de 959328 bytes.

A continuación se muestran algunos de los resultados obtenidos:

Resumen de la prueba:

#Samples	KO	Error %	Average response time	90th pct	95th pct	99th pct	Throughput	Received KB/sec	Sent KB/sec
7500	**0**	**0.00%**	**6729.79**	**13323.60**	**15556.90**	**19229.87**	**12.25**	**11480.48**	**2.92**
7500	0	0.00%	6729.79	13323.60	15556.90	19229.87	12.25	11480.48	2.92

Tabla 2. Resumen de la prueba.

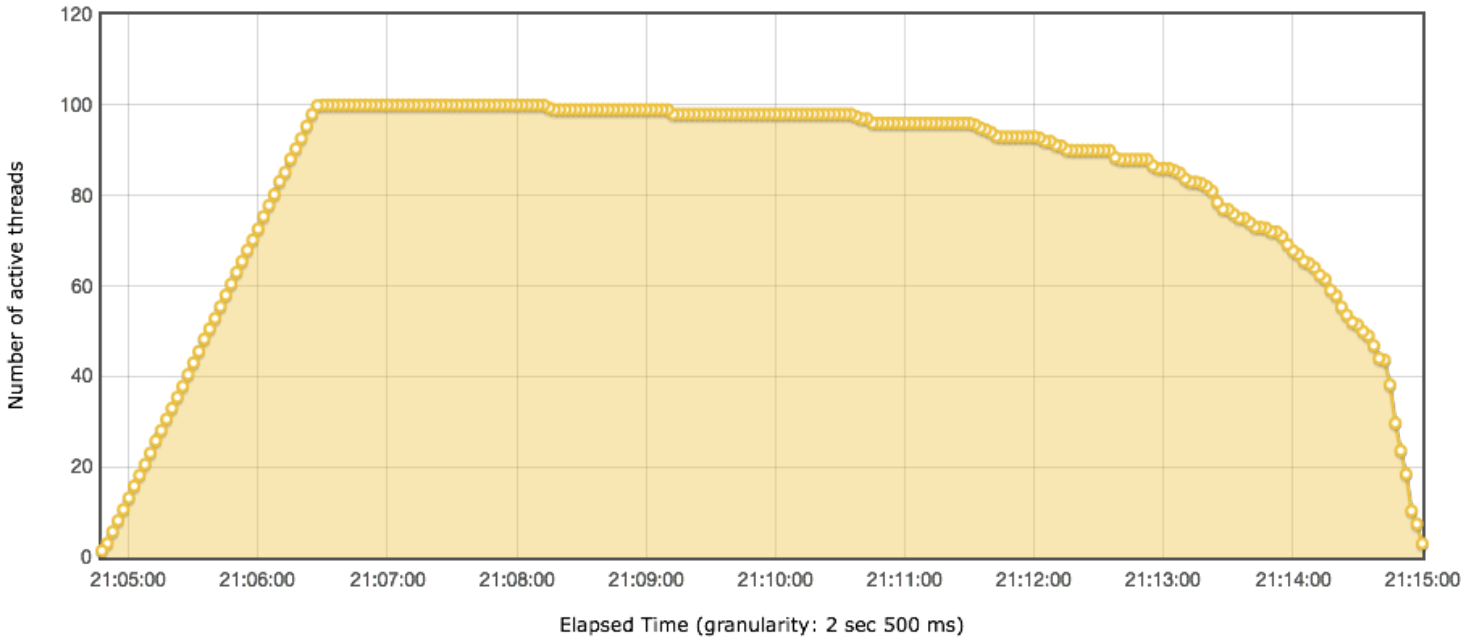


Figura 1. Threads activos en función del tiempo.

Durante los primeros 100 segundos, se fue creando un nuevo thread por segundo, hasta llegar a los 100. Después de ese punto, la cantidad de threads se mantuvo constante mientras se ejecutaban los 75 loops de la prueba.

En las *figuras 2 y 3* se puede ver claramente cómo la latencia y el tiempo de respuesta se corresponden con la cantidad de threads que estuvieron activos a lo largo de la prueba. En la *figura 4*, de tiempo de respuesta promedio en función de los threads activos, se puede ver que la línea de tendencia tiene forma lineal.

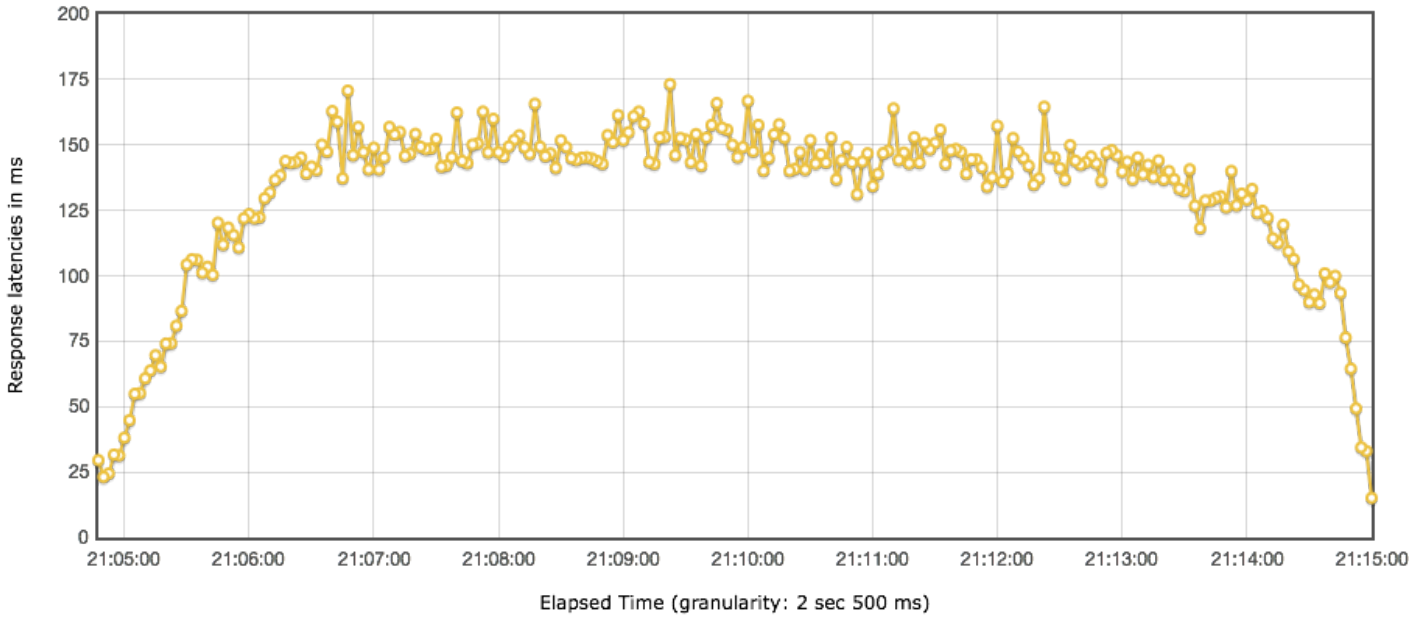


Figura 2. Latencia en función del tiempo.

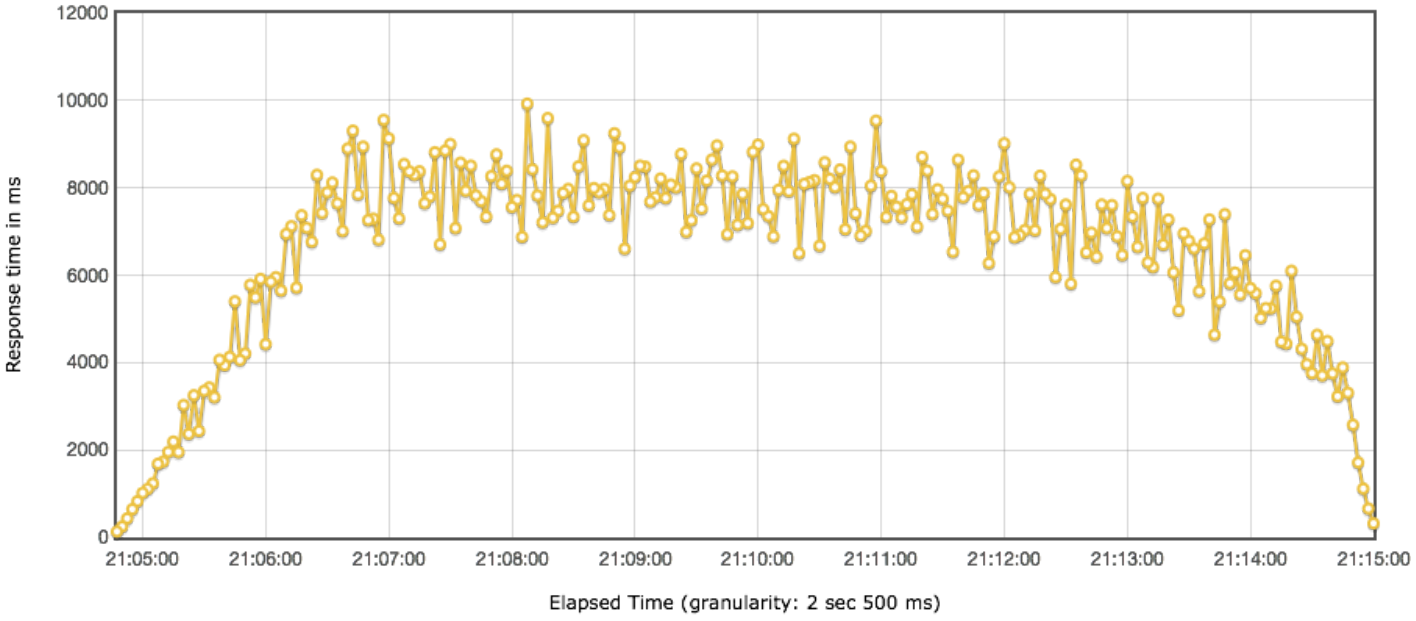


Figura 3. Tiempo de respuesta en función del tiempo.

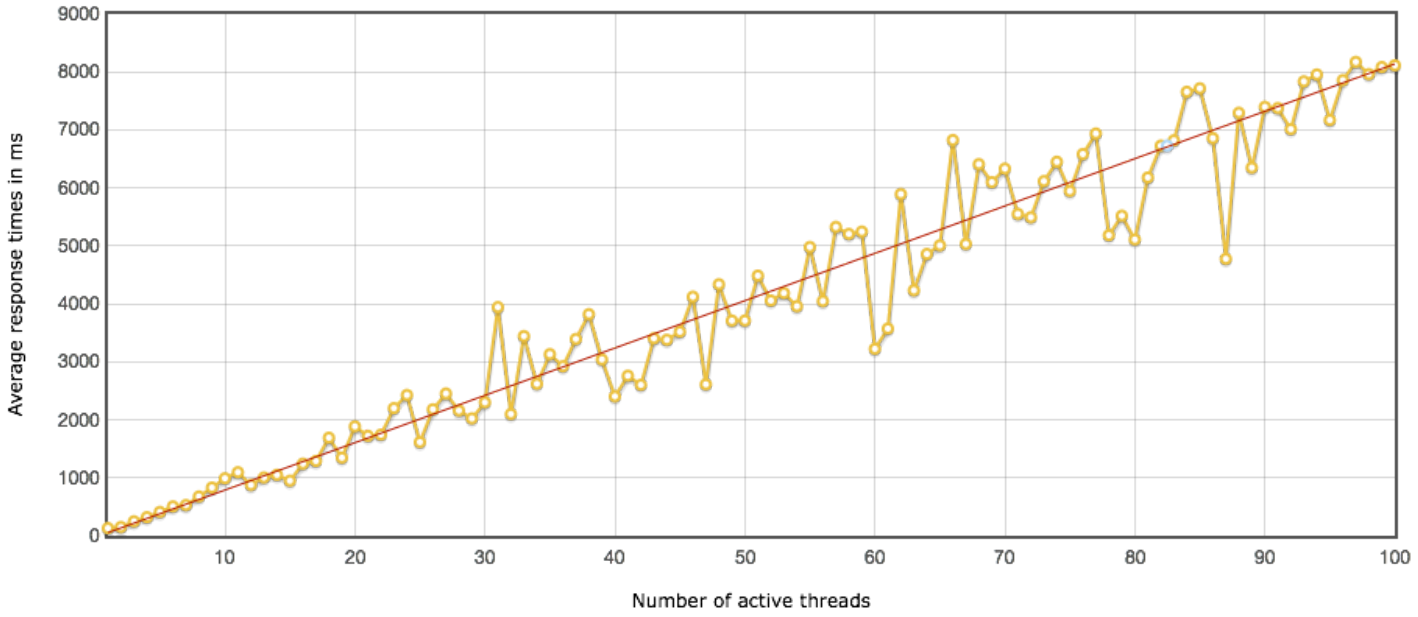


Figura 4. Tiempo de respuesta promedio en función de los threads activos.

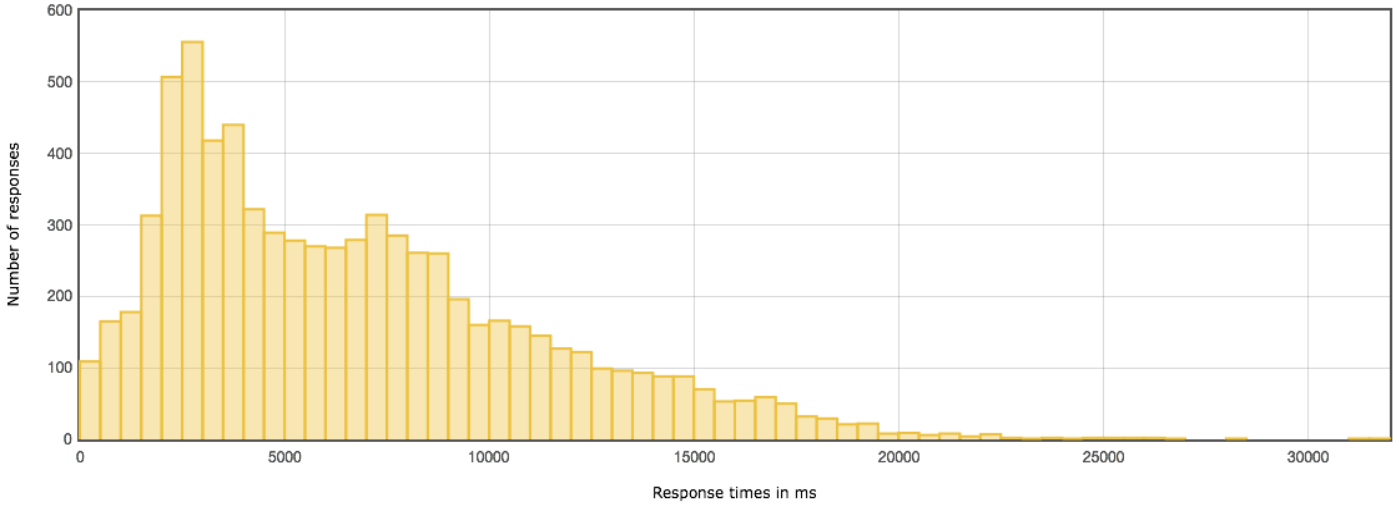


Figura 5. Histograma de tiempos de respuesta.

La mayoría de las respuestas tuvieron un tiempo de respuesta de alrededor de 2500 ms, mostrando la estabilidad del proxy una vez alcanzada la cantidad máxima de threads de la prueba. Lo mismo puede verse en la curva de *figura 6,* de percentilos de tiempos de respuesta, la cual tiene una pendiente suave, indicando que no hay una varianza significativa en los tiempos de respuesta.

En el único punto donde crece de manera más abrupta es en el primer percentil, para tiempos de respuesta bajos. Esto se corresponde con la etapa de ramp-up de la prueba, cuando había pocos threads y por ende, un menor tiempo de respuesta, lo que rápidamente cambió JMeter fue creando más hilos.

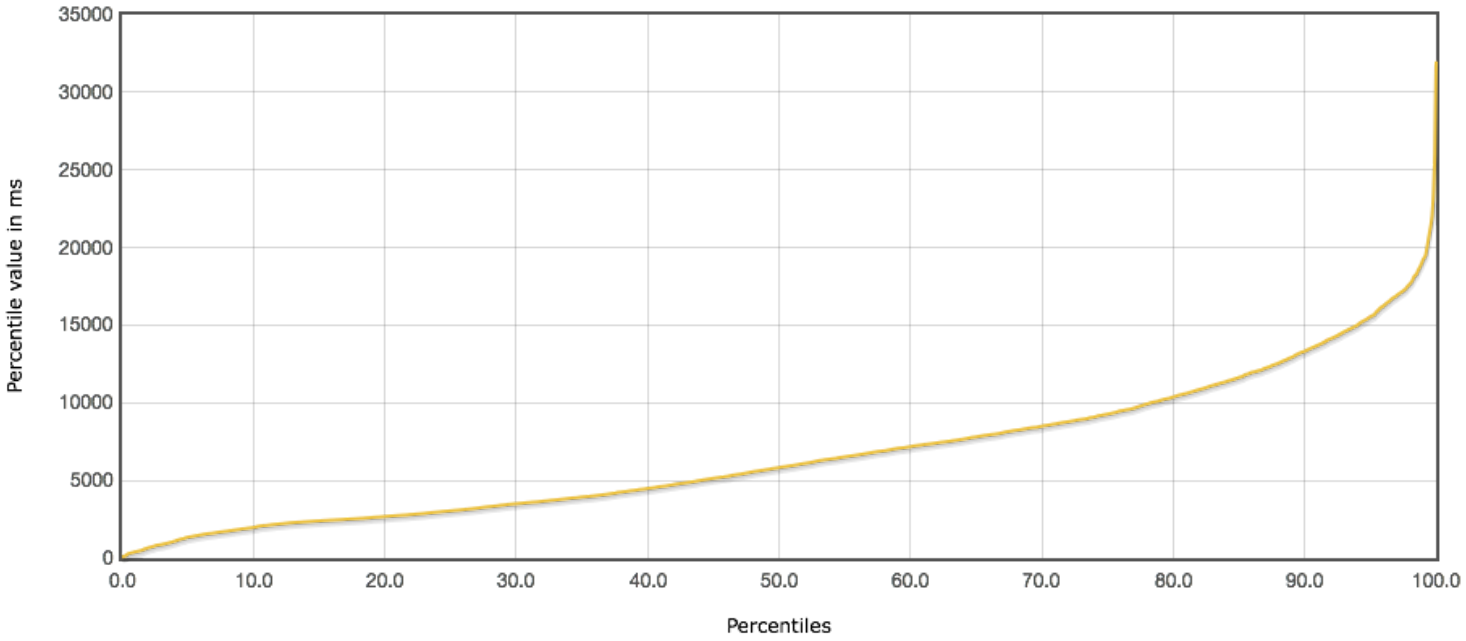


Figura 6. Percentilos de tiempo de respuesta.

Pruebas de Funcionalidad

L33t charset válido

Pre-condiciones:

- User-agent: cURL configurado para que los requests pasen por el proxy
- L33t se encuentra prendido
- Se encuentra corriendo el servidor provisto por la cátedra en **foo**
- Se aplicó el parche provisto por la cátedra

Pasos a seguir:

```
$ curl -s -i http://foo/api/chunked/leet/ascii

$ curl -s -i http://foo/api/chunked/leet/utf-8

$ curl -s -i http://foo/api/chunked/leet/iso-8859-1
```

Post-condiciones:

- Se observa la transformación y 5000 líneas (no hay pérdida) en todos los casos

POST sin Content-Length ni Transfer-Encoding: chunked

Pre-condiciones:

- Netcat sin configuración en especial
- **proxy** es el IP/hostname donde corre el proxy y **port** el puerto del mismo

Pasos a seguir:

```
$ nc -C proxy port
$ POST /api/other HTTP/1.1
$ Host: foo
$
```

Respuesta:

```
HTTP/1.1 411 Length Required
Content-type: text/plain
Connection: close
Content-length: 88

411 Length Required: Missing valid content-length and transfer-encoding:
   chunked headers
```

Post-condiciones:

- La respuesta debe dar código de error 411
- Se cierra la conexión luego de enviar los headers

Petición sin host

Pre-condiciones:

- Netcat sin configuración en especial
- **proxy** es el IP/hostname donde corre el proxy y **port** el puerto del mismo

Pasos a seguir:

```
$ nc -C proxy port
$ GET / HTTP/1.1
$ X-header: irrelevant
$
```

Respuesta:

```
HTTP/1.1 400 Bad Request
Content-type: text/plain
Connection: close
Content-length: 48

400 Bad Request: Missing host in headers and URL
```

Post-condiciones:

- La respuesta debe ser como la indicada
- Se cierra la conexión luego de enviar los headers

Guia de instalación y configuración

Ubicación de Archivos

En el directorio raíz se encuentra el directorio **proxy-http** que contiene el código fuente como también el archivo de construcción pom.xml. En el directorio raíz se encuentra también el informe con el nombre informe.pdf y el archivo de presentación presentación.pdf.

Construcción y ejecución

Ejecutar el comando `mvn package` dentro del directorio proxy-http. Se generará el directorio **proxy-http/target** donde en el directorio **proxy-http/target/lib** se encuentran las dependencias. Para correr la aplicación, ejecutar el comando `java -jar proxy-http-1.0.java` dentro del directorio **proxy-http/target**. En el directorio **proxy-http/target/logs** se generarán los archivos de logs.

Configuración

Para configurar puertos y tamaño de buffers, se puede abrir el archivo proxy-http/target/proxy-http-1.0.java con un editor de textos como vim y modificar el archivo de configuración proxy.properties. Se describen a continuación las diferentes configuraciones:

- **proxy.port**: puerto de la aplicación proxy
- **proxy.bufferSize**: tamaño default de buffers de transferencia del proxy
- **protocol.port**: puerto de la aplicación del protocolo
- **protocol.bufferSize:** tamaño de buffers de transferencia del protocolo
- **protocol.parser.bufferSize:** tamaño de buffer del parser del protocolo
- **protocol.parser.headerNameBufferSize:** tamaño de buffer para nombre del header del protocolo
- **protocol.parser.headerContentBufferSize:** tamaño de buffer para el contenido de un header
- **parser.methodBufferSize:** tamaño del buffer para el método HTTP
- **parser.requestLineBufferSize:** tamaño del buffer de primer línea de la request HTTP
- **parser.URIHostBufferSize:** tamaño del buffer que guarda host si se encuentra en la primer línea
- **parser.headerNameBufferSize:** tamaño del buffer del nombre de headers HTTP
- **parser.headerContentBufferSize:** tamaño del buffer del contenido de headers HTTP
- connection.queue.length: cantidad de conexiones que se persisten de un mismo par host:puerto
- **connection.ttl:** time to live en segundos de una conexión persistida
- **connection.clean.rate:** cada cuantos segundos se realiza una limpieza de conexiones cuyo ttl expiró

Monitoreo

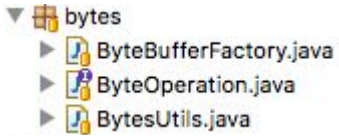
Para el monitoreo de la aplicación se proveen logs de acceso como también logs de error. Se genera uno nuevo cada día hasta un máximo de 5 logs para un mismo tipo de log. El formato de los mismos es el detallado previamente.

A su vez, puede monitorearse el estado del proxy a partir de los métodos de métricas provistos por el protocolo.

Diseño y Arquitectura del Proyecto

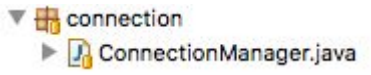
Se presenta a continuación la arquitectura de los paquetes, seguido de una explicación de los contenidos de cada uno.

Bytes



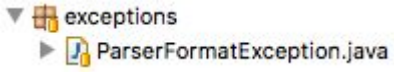
Provee clases de utilidad para el manejo de bytes como también un Factory para la creación de los buffers del proxy. Permite que se modifique el tamaño de los mismos en runtime.

Connection



Se encarga de manejar las conexiones con los servidores HTTP, utilizando una conexión cacheada o estableciendo la conexión según sea necesario.

Exceptions

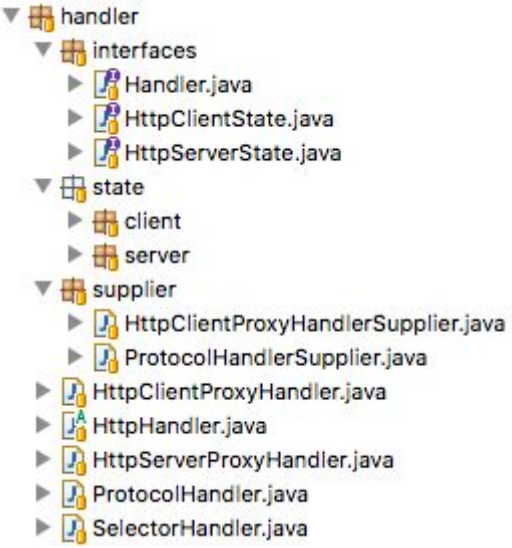


Cachable exception que indica un error sintáctico de parseo, ya sea al parsear HTTP o el comandos del protocolo.

Flag

Singleton que representa el estado del flag de transformaciones l33t. Provee métodos para consultar y modificar dicho estado.

Handler

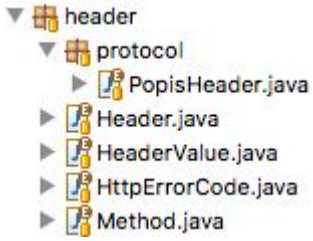


Contiene los handlers que se encargan se ejecutan la debida acción luego de recibir una key lista para alguna operación.

En el paquete **interfaces** se presentan interfaces que implementan los estados y los handlers.

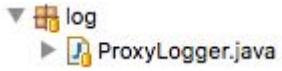
En el paquete **state** se encuentran las clases de estados correspondientes a `HttpServerProxyHandler` y `HttpClientProxyHandler`.

Header



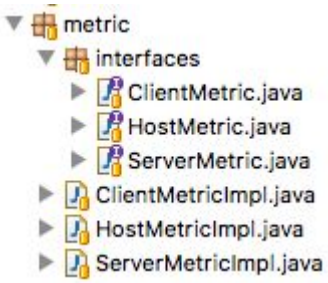
Presenta Enums con headers del protocolo, headers HTTP, contenido de headers HTTP, métodos HTTP y Status Codes de la familia 400/500 HTTP.

Log



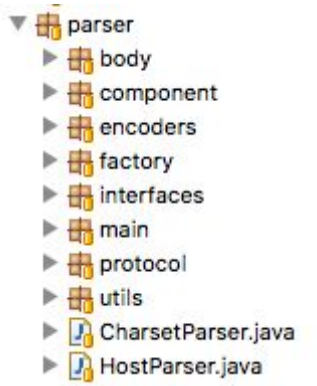
Singleton que se encarga de loguear a disco los registros de acceso y de error.

Metric



Contiene singletons que manejan las métricas del cliente y servidor, como también interfaces que disponen de dichos métodos.

Parser



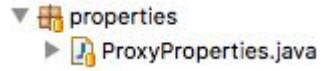
Contiene todas las clases que conciernen a cualquier actividad de parseo como también la creación de objetos parsers.

En el paquete **main** se encuentran los parsers principales de una request y response HTTP, compuestos por parsers de los paquetes **body** y **component**.

En el paquete **interfaces** se presentan las interfaces que implementan todos estos parsers.

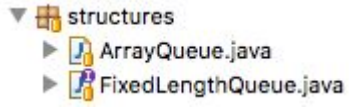
En el paquete **protocol** se presentan los parsers del protocolo.

Properties



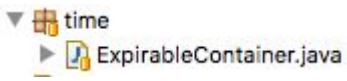
Singleton que se encarga de leer el archivo de configuración

Structures



Estructura que representa una cola de tamaño fijo, acompañada por la interfaz correspondiente.

Time



Wrapper para un objeto que puede expirar a partir de un timeout.