Protocolos de comunicación

Informe final

- Besteiro, Florencia 51117
- Buireo, Juan Martín 51061
- Menzella, Facundo 51533
- Purita, Martín 51187

14-11-2013

Índice

Objetivo

Implementar un servidor proxy para el protocolo HTTP versión 1.1 (Hypertext Transfer Protocol) [RFC2616] que pueda ser usado por User agents como Mozilla Firefox, Internet Explorer y Google Chrome para navegar por Internet.

Características del servidor proxy a implementar

- Debe ser un proxy transparente.
- También debe poder ser usado desde los User Agents con configuración previa de los mismos.
- Debe soportar múltiples clientes de forma concurrente y simultánea.
- Debe poder configurarse para que realice sus requests hacia el origin server a través de otros servidores proxies (encadenamiento de proxies).
- ***Opcional:*** El proxy PUEDE soportar conexiones persistentes (RFC 2616 sección 8.1.1) desde los clientes y hacia los servidores (no es requerido soportar pipelining).

En nuestro caso, debido a tener 4 integrantes, debimos cumplir con el requerimiento opcional (Conexiones persistentes).

Descripción de los protocolos y aplicaciones desarrolladas

Sockets

El proxy se desarrolló utilizando sockets no bloqueantes, particularmente Nio.

Se eligió esta opción frente a la de bloqueantes ya que los sockets no bloqueantes son más performantes en servidores concurrentes.

Por otro lado, debido a que la implementación de conexiones persistentes era de carácter obligatorio, nos inclinamos aún más por esta opción.

Protocolo de monitoreo

Como el trabajo práctico se basa en un proxy HTTP (versión 1.1), se decidió implementar el protocolo de monitoreo sobre HTTP debido a que ya se tenía implementado un parser con lo cual bastó sólo con extenderlo. De esta manera se ahorró trabajo y se minimizó la probabilidad de error al contar con un solo protocolo en la aplicación.

Por otro lado, analizando las características de HTTP, se llegó a la conclusión que nuestro protocolo podía basarse en el mismo. En primer lugar, HTTP es un protocolo request/response lo cual se adapta perfectamente a nuestro protocolo ya que se solicita una acción y se obtiene una respuesta de la misma con sus respectivos datos. En segundo lugar, HTTP no mantiene estado, lo cual es algo que tampoco debemos mantener en nuestro protocolo.

Otra ventaja de utilizar HTTP es que se pudieron reutilizar ciertos headers ya provistos, como es el caso de Accept (para la negociación de contenido entre cliente y servidor) o de Authorization (para evitar que cualquiera pueda modificar o consultar el estado del proxy).

Se reutilizaron solamente dos métodos: GET y POST.

El método GET se utilizó para aquellos comandos de consulta, por ejemplo, bytes transferidos o para pedir el histograma de status codes.

El método POST se utilizó para aquellos comandos de configuración. Se hizo de esta manera ya que este método no es idempotente, es decir, no produce el mismo

resultado si se ejecuta una o varias veces. Y esto tiene sentido ya que se está modificando una configuración interna del proxy. En nuestro caso, sólo hay una posible configuración que consiste en activar o desactivar la transformación de mensajes.

Se definieron los siguientes tipos de cuerpo de respuesta:

application/vnd.ehttp-histogram

Respuesta = c : { (h : v) }
c = código solicitado definido en la rfc 2616, sección 10 (Ejemplo: 200)
h = 0 | 1 | 2 | … | 22 | 23
v = 0 | 1 | 2 | … | N (siendo N un número entero representando la cantidad de repeticiones para el status solicitado)
Se entiende que la tupla h : v, se encuentra repetida 24 veces, correspondiente a cada hora del día.

application/vnd.ehttp-transformer

Respuesta = { "TRANSFORMER" : s }
s = "Disabled" | "Enabled"

application/vnd.ehttp-accesses

Respuesta = { "Accesses" : s }
s = [0-9]+

application/vnd.ehttp-status

Respuesta = { "TRANSFORMER" : s }
s = "Disabled" | "Enabled"

application/vnd.ehttp-bytes

Respuesta = { "Bytes read" : br , "Bytes total tranfered" : btt, "Bytes written" : bw,"Bytes changed" : bc}
Br = btt = bw = bc = [0-9]+
Se entiende que este es un formato para el cuerpo de respuesta, y que los headers http correspondientes son reutilizados.

Problemas encontrados durante el diseño y la implementación

A lo largo de la implementación surgieron varios problemas y varias discusiones las cuales fueron aclaradas con los profesores.

Parseo a nivel de String

Debido a que se debía parsear tanto el request (del cliente hacia el origin server) como el response (del origin server hacia el cliente) para analizar headers y el body (en caso que sea chunked o la transformación de mensajes estuviese activada), era necesario convertir el contenido del ByteBuffer (que obviamente guarda de a bytes), a algún formato legible. Fue por eso que se tomó la decisión de convertirlo a String.

Esto nos trajo problemas con ciertos requests que se realizaban, como es el caso de solicitar una imagen. Debido a que la misma tiene caracteres extraños, al convertirlos a String se malinterpretaban dichos caracteres con lo cual la imagen quedaba deformada o modificada. La solución fue manejar esto con parseo a nivel de bytes.

Transfer-Encoding

Uno de los problemas que nos surgió fue el de la compresión. Observamos que algunos origin server nos enviaban su respuesta (el body) comprimida con lo cual nos dificultaba la lógica en caso que se nos enviara un text/plain (ya que habría que hacer los cambios correspondientes). Por otro lado, no manejábamos la compresión.

Ante esto surgieron dos soluciones posibles:

- En caso de recibir una respuesta comprimida, reenviar nuevamente el request al origin server solicitándole una respuesta sin compresión
- Eliminar el header Accept-Encoding del request para asegurarnos de esta manera que la respuesta esta descomprimida

Lógicamente optamos por la segunda opción ya que no tenía sentido recibir algo que no sorportabamos y comunicárselo luego al origin server. Era mucho más sencillo directamente eliminar aquello que no éramos capaces de resolver.

Otro problema que surgió fue con el “chunked”. Como primera opción se nos ocurrió intentar pedirle al origin server que no mande la respuesta en formato chunked, tal

como se hizo con el contenido comprimido. Luego de leer el rfc, se encontró que con el header *TE: "identity;q=1, chunked;q=0"* se podía deshabilitar el chunked. Pero luego de varias pruebas se llegó a la conclusión que muchos origin servers ignoraban dicho header y enviaban la respuesta en formato chunked.

Fue allí cuando se decidió que no quedaba otra alternativa que implementarlo. Para ello, nuevamente se leyó la sección correspondiente en el rfc para comprender su funcionamiento.

Multithread

Nuestro proxy es actualmente monothread.

Hubo un intento de realizar nuestro proxy en múltiples threads (para poder realizar procesamiento en paralelo) pero eso no fue posible debido a que al hacerlo, surgieron problemas en las "keys" de Nio. Esto sucede ya que el set de "keys" no es thread safe con lo cual puede pasar que una clave este siendo utilizada y luego se la intente remover del mismo.

Conexiones persistentes

Para la implementación de conexiones persistentes hubo una discusión muy importante acerca de entre quienes se debía mantener la persistencia. Había dos opciones posibles:

- Mantener la persistencia entre cliente y origin-server
- Mantener la persistencia sólo entre proxy y origin-server

La opción correcta era la segunda ya que lo que interesaba era mantener abierto un canal con cierto origin-server para poder volver a solicitarlo.

Luego surgió una segunda discusión acerca de cómo debía ser dicha conexión persistente entre el proxy y el origin-server. Dicha discusión se reduce a los siguientes esquemas:

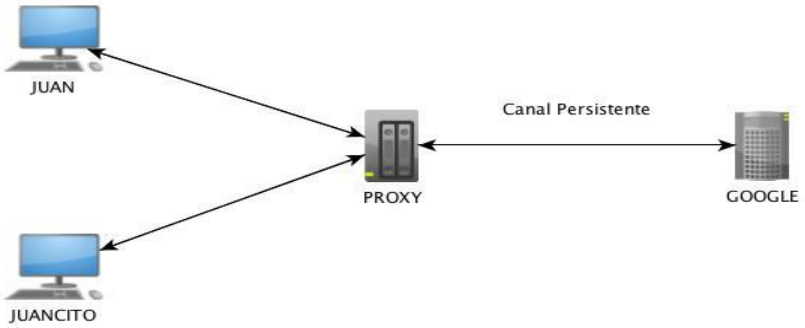

Figura 1: única conexión persistente entre proxy y origin server

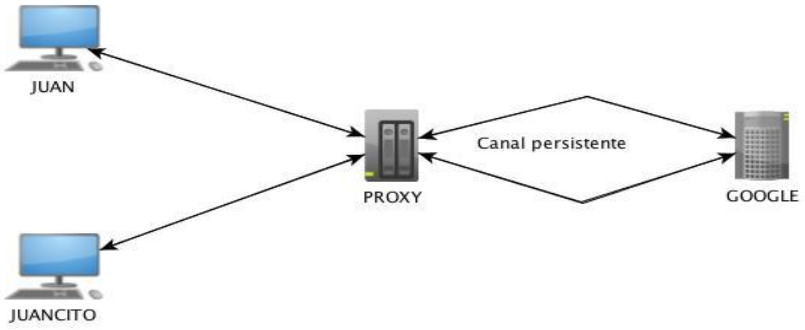

Figura 2: una conexión persistente por cada cliente entre proxy y origin server

En la figura 1 se puede observar que a pesar de haber dos clientes solicitando el mismo origin server (Google), ambos pasan por el mismo canal persistente. Esta opción se descartó ya que podríamos tener problemas de concurrencia y la solución sería que un cliente espere al otro. Obviamente esta opción no se consideró y se optó por el esquema de la figura 2. Allí se puede observar que si dos clientes (o más) solicitan al mismo tiempo el mismo origin server con conexión persistente, se abrirá una conexión por cada uno.

Para mantener las conexiones, se guardó en un mapa (ConcurrentMap) dichas conexiones de manera que si se solicita una conexión persistente basta con buscarla en el mapa y si la misma no se encuentra, se la crea. En el caso previamente explicado de que muchos clientes soliciten al mismo tiempo el mismo origin server, al momento de guardar dichas conexiones, se mantienen un cierto número en el mapa y no todas.

HTTPs

En etapas tempranas de la implementación del proxy, se pensó que el mismo andaba correctamente (sin haber implementado muchos puntos que dice el rfc) ya que se probaba contra páginas HTTPs como Google, Facebook, etc.

Lo que nos llevó a pensar esto fue que al ver la consola del Eclipse, se veía flujo de datos. Con el correr del tiempo comprendimos que lo único que se veía era el redirect que se realizaba hacia las páginas HTTPs. Una vez hecho el mismo ya no se pasaba por el proxy con lo cual podíamos navegar perfectamente.

Flujo de datos

El principal problema que tuvimos con el flujo de datos es que inicialmente almacenábamos toda la información del request y response en memoria lo cual solamente nos permitía navegar por páginas de pequeño tamaño. Cuando intentábamos descargar la iso de ubuntu el proxy dejaba de funcionar por insuficiencia de heap lo cual nos llevó a la conclusión de cambiar de implementación.

Decidimos en caso de tener un content-length mayor a 10 Mb o cuando el response tiene transfer-encoding: chunked bajar el body a un archivo temporal el cual se borra luego de enviarle al cliente por el socket dicho archivo.

Esta implementación nos permitió fácilmente la mejora del proxy y la posibilidad de descargar la iso de ubuntu así como también navegar por páginas de gran tamaño.

Limitaciones de la aplicación

Compresión de datos

Actualmente, el proxy no soporta archivos gzipeados. Esto le saca mucha performance, dado que el volumen de datos a transferir es exponencialmente mayor a el tamaño que tendrían si estuviesen comprimidos.

Posibles extensiones

Una posible extensión que mejoraría muchísimo la performance sería implementar la descompresión de datos. Al analizar el tráfico (sin proxy) entre una respuesta comprimida y una sin comprimir se aprecia una diferencia importante en el flujo de datos recibidos.

Otra posible extensión y que puede ser útil tratándose de un proxy sería que mediante el protocolo, se pueda bloquear el acceso a ciertas páginas no deseadas. Debido a la facilidad del protocolo y a que está basado sobre HTTP sería bastante simple de implementar. Habría que hacerlo mediante POST y bastaría con mandar en algún header la página que se desea bloquear.

Otra posible extensión sería implementar caché. Esto mejora considerablemente la performance pero también entendemos que no es algo sencillo de implementar (y en este caso no quedaría otra que conservar datos en disco).

Conclusiones

Implementar un proxy HTTP no es una tarea sencilla. Hay que tener muchas cosas en cuenta antes de intentar realizarlo. La lectura de los rfcs apropiados facilitan la tarea.

Un proxy bien implementado puede ser muy útil ya que puede reducir el flujo de datos o bien recolectar ciertas estadísticas que pueden servir para un análisis posterior. Por otro lado, puede ser muy útil para el bloqueo de ciertas páginas no deseadas.

Ejemplos de testeo

Test1: Apache server benchmarking tool

Servidor nginx en máquina remota (rfc 2616) – 5 requests concurrentes

Comando:

```
ab -n 5 -c 5 -X 192.168.0.106:9090 http://192.168.0.108:8080/
```

Concurrency Level: *5*
Time taken for tests: *16.698 seconds*
Complete requests: *5*
Failed requests: *0*
Write errors: *0*
Total transferred: *2112575 bytes*
HTML transferred: *2111395 bytes*
Requests per second: *0.30 [#/sec] (mean)*
Time per request: *16698.487 [ms] (mean)*
Time per request: *3339.697 [ms] (mean, across all concurrent requests)*
Transfer rate: *123.55 [Kbytes/sec] received*

Percentage of the requests served within a certain time (ms)
50% 15036
66% 16479
75% 16479
80% 16698
90% 16698
95% 16698
98% 16698
99% 16698
100% 16698 (longest request)

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	0 0.1	0	0
Processing:	14363	15521 1014.7	15757	16698
Waiting:	14363	15521 1014.7	15757	16698
Total:	14363	15522 1014.7	15757	16698

Servidor nginx en máquina local (rfc 2616) – 5 requests concurrentes

Comando:

ab -n 5 -c 5 -X 192.168.0.106:9090 *http://localhost:8080/*

Concurrency Level: *5*
Time taken for tests: *2.381 seconds*
Complete requests: *5*
Failed requests: *0*
Write errors: *0*
Total transferred: *2112575 bytes*
HTML transferred: *2111395 bytes*
Requests per second: *2.10 [#/sec] (mean)*
Time per request: *2380.523 [ms] (mean)*
Time per request: *476.105 [ms] (mean, across all concurrent requests)*
Transfer rate: *866.64 [Kbytes/sec] received*

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	0	0.0	0
Processing:	2375	2378	2.1	2379
Waiting:	2375	2378	2.1	2379
Total:	2376	2378	2.0	2380

Percentage of the requests served within a certain time (ms)

50% 2379
66% 2380
75% 2380
80% 2380
90% 2380
95% 2380
98% 2380
99% 2380
100% 2380 (longest request)

Guías de instalación

Desde Eclipse

1. El archivo de configuración del proxy se encuentra en src/main/resources y se llama proxy.properties.
2. Los logs se encuentran en la carpeta log.
3. Para ejecutar el proxy server basta con ejecutar la clase TCPServerSelector.

Generando el jar

1. Situarse donde se encuentra el archivo pom.xml.
2. Allí desde una consola ejecutar el siguiente comando: mvn clean package. Esto generará el jar correspondiente.
3. Finalizado el paso anterior ingresar a la carpeta generada (llamada target).
4. El archivo de configuración del proxy se encuentra en clases y se llama proxy.properties.
5. Los logs se encuentran en la carpeta log.
6. Para ejecutar el proxy server basta con situarse en la carpeta target y ejecutar el siguiente comando: java –jar HTTPProxy-withdependencyes.

Desde un User Agent con configuración previa

1. Abrir el archivo proxy.properties y configurar la IP local.
2. Abrir el user agent deseado e ir a configuración de proxy.
3. Una vez allí activar la opción de usar un servidor proxy.
4. Introducir la IP local y el puerto 9090.
5. Correr el proxy server.
6. Comenzar a navegar por la web (sin usar páginas https).

Desde netcat

1. Ejecutar el proxy server.
2. Ejecutar una terminal e introducir nc [página solicitada] 9090
3. Introducir GET / HTTP/1.1
4. Introducir Host: [página solicitada]
5. Introducir algún otro header que se quiera enviar.

Encadenamiento de proxies

1. Abrir el archivo proxy.properties y configurar el server-ip con la dirección IP local (es decir donde se está coriendo el proxy).
2. Configurar el chained-ip con la dirección IP del proxy al cual se desea encadenarse.
3. Configurar el chained-port con el puerto en el cual está corriendo el proxy al que se desea encadenar.
4. Continuar con alguno de los casos anteriores

Instrucciones para la configuración

Desde Eclipse

Para configurar el proxy desde Eclipse basta con abrir el archivo proxy.properties (que se encuentra en src/main/resources). Allí se pueden configurar tanto el puerto del proxy como el puerto del protocolo de administración y la IP local desde donde está corriendo el proxy server.

Desde allí también se pueden configurar los parámetros para el encadenamiento de proxies (la dirección IP y el puerto del proxy al cual se desea encadenar).

Desde el jar generado

Para configurar el proxy desde Eclipse basta con abrir el archivo proxy.properties (que se encuentra en target/classes). Allí se pueden configurar tanto el puerto del proxy como el puerto del protocolo de administración y la IP local desde donde está corriendo el proxy server.

Desde allí también se pueden configurar los parámetros para el encadenamiento de proxies (la dirección IP y el puerto del proxy al cual se desea encadenar).

Ejemplos de configuración y monitoreo

Ejemplo 1:

GET /bytes HTTP/1.1
Host: ehttp
Authorization: Basic ZWh0dHA6MTIzNAo=

HTTP/1.0 200 OK
Date: 2013/11/13 16:20:41
Content-Length: 120
Connection: close
Cache-Control: no-cache

{ "Bytes read" : "21554310", "Bytes total tranfered" : "29585422", "Bytes written" : "8031112", "Bytes changed" : "0" }

Ejemplo 2:

GET /histogram?code=200 HTTP/1.1
Host: ehttp
Authorization: Basic ZWh0dHA6MTIzNAo=

HTTP/1.0 200 OK
Date: 2013/11/13 16:23:03
Content-Length: 282
Connection: close
Cache-Control: no-cache

{ "0" : "0", "1" : "0", "2" : "0", "3" : "0", "4" : "0", "5" : "0", "6" : "0", "7" : "0", "8" : "0", "9" : "0", "10" : "0", "11" : "0", "12" : "0", "13" : "0", "14" : "0", "15" : "0", "16" : "70", "17" : "0", "18" : "0", "19" : "0", "20" : "0", "21" : "0", "22" : "0", "23" : "0" }

Ejemplo 3:

GET /status HTTP/1.1
Host: ehttp
Authorization: Basic ZWh0dHA6MTIzNAo=

HTTP/1.0 200 OK
Date: 2013/11/13 16:26:10
Content-Length: 31
Connection: close
Cache-Control: no-cache

{ "TRANSFORMER" : "Disabled" }

Ejemplo 4:

POST /transformer HTTP/1.1
Host: ehttp
Authorization: Basic ZWh0dHA6MTIzNAo=
Content-Length: 0

HTTP/1.1 200 OK
Date: 2013/11/13 16:27:10
Content-Length: 30
Connection: close
Cache-Control: no-cache

{ "TRANSFORMER" : "Enabled" }

Ejemplo 5:

GET /accesses HTTP/1.1
Host: ehttp
Authorization: Basic ZWh0dHA6MTIzNAo=

HTTP/1.0 200 OK
Date: 2013/11/13 16:32:49
Content-Length: 22
Connection: close
Cache-Control: no-cache

{ "Accesses" : "25" }

Diseño del proyecto