Proxy HTTP 1.1

Informe de desarrollo

Matías Ezequiel Colotto
María Eugenia Cura
Santiago José Samra
Jorge Ezequiel Scaruli

Protocolos de Comunicación
2010

Índice

1. Introducción

Se propuso como Trabajo Práctico Especial de la materia Protocolos de Comunicación el desarrollo de un servidor proxy para el protocolo *HTTP (Hypertext transfer protocol)* en su versión 1.1, que pueda ser utilizado para navegar en Internet y que sea compatible con los navegadores *Mozilla Firefox*, *Google Chrome* e *Internet Explorer*.

El objetivo de este informe es dar a conocer las funcionalidades implementadas y describir el desarrollo de las mismas. Además, se presentan pruebas de carga realizadas al servidor, así como posibles extensiones de la implementación.

2. Protocolos desarrollados

El proyecto se realizó completa y únicamente sobre el protocolo *HTTP* 1.1 especificado por la RFC 2616[1]. Por lo tanto, no se desarrollaron ni implementaron protocolos propios.

El servidor ofrece la posibilidad de ser configurado y monitoreado remotamente y, aunque esto pudo haber sido implementado mediante protocolos propios, se decidió que no sea así. Una de las razones de ello fue la posibilidad de utilizar un protocolo ya existente como *HTTP*, con lo que se evitó la dificultad de diseñar uno propio teniendo en cuenta las dificultades que implica. Otra razón fue el hecho de que, al usar *HTTP*, es posible ofrecer usabilidad web para estos servicios. Finalmente, el hecho de poder aprovechar las características de parseo de los *requests HTTP*, que fueron desarrolladas para el funcionamiento del *proxy*, hizo que implementar el monitoreo y la configuración sea relativamente simple.

[1] http://tools.ietf.org/html/rfc2616

3. Problemas encontrados

Durante el desarrollo se encontraron gran cantidad de problemas, que debieron ser solucionados para lograr un producto sólido. Se mencionan a continuación algunos de ellos:

3.1. Decisión sobre el uso de *threads* o E/S asincrónica

En un principio, para ponerse en contacto con las funcionalidades que ofrece Java para programar servidores concurrentes y manejo de *threads*, se procedió a implementar pequeños servidores que ofrezcan servicios básicos (como por ejemplo, *echo*). Estas implementaciones se hicieron utilizando las dos alternativas analizadas en la materia:

1. Teniendo un servidor que, por cada pedido de un cliente, lance un nuevo *thread* que lo atienda.
2. Mediante entrada/salida asincrónica (utilizando el mecanismo *select*), en el cual los pedidos entrantes de los clientes se mantienen inactivos hasta que el servidor los tome y los atienda.

Analizando el funcionamiento de cada una de las dos alternativas, se observó que la primera tenía un funcionamiento más acorde a lo que se esperaba del servidor ya que, básicamente, la segunda opción tenía un modo de trabajar basado en un solo thread; entonces, más allá de soportar que varios pedidos de clientes se hagan simultáneamente, no era posible que esos pedidos se atendieran al mismo tiempo.

Sin embargo, se pensó en una tercera alternativa que consistía en una combinación de las dos anteriormente mencionadas. Consistía en que el servidor conste de tres *threads*:

- Uno que espere conexiones entrantes, y las coloque en una lista de conexiones a monitorear por el mecanismo *select*.
- Uno que se encargue de realizar las operaciones de lectura en las conexiones que se sacaban de dicha lista.
- Otro que realice las operaciones de escritura.

Con esto, se lograba un funcionamiento más eficiente del mecanismo *select*, porque no era necesario que el servidor esté desocupado para atender una conexión entrante; era posible que un thread acepte conexiones mientras otro las atendía. Asimismo, se evitaba el lanzamiento de un nuevo *thread* por cada conexión, algo que implicaba un gran consumo de memoria y procesamiento.

No obstante, la tercera opción no fue llevada a la práctica, debido a que se intentó programar un servidor *echo* implementándola y se observó que el funcionamiento ideal implicaba un gran control de sincronización entre los *threads*, algo que difícilmente fue logrado en el servidor *echo* luego de muchos intentos. Por lo tanto, intentar aplicarlo en el proxy llevaría a grandes complicaciones.

Finalmente, se decidió implementar una variante de la primera opción mencionada, en la cual en vez de disparar un nuevo *thread* por cada pedido de un cliente, se tome uno entre varios disponibles de un *pool*. De esta manera, se limitó la cantidad de *threads* existentes al mismo tiempo en el servidor. Aunque esta limitación implica una menor cantidad de *threads* funcionando simultáneamente en el servidor, hace que el consumo de memoria sea moderado y que no se requiera tanta capacidad de procesamiento. De todos modos, la cantidad de *threads* presentes en el *pool* puede ser configurada mediante el archivo de configuración del proxy, con lo cual estas limitaciones pueden aplicarse en función de las características de la máquina que sirva el proxy y así aprovecharlas para un funcionamiento óptimo.

3.2. Interpretación y parseo de mensajes *HTTP*

Luego de la decisión tomada acerca de cómo atender clientes, se llevó a cabo el parseo de mensajes *HTTP*.

Antes de diseñar los *parsers*, se pensó una jerarquía de clases que modele los paquetes *HTTP*[2]. A grandes rasgos, consistió en una clase padre `HttpPacket` de la cual heredan `HttpRequest` y `HttpResponse`, y que contienen un método (`GET`, `POST` o `HEAD`, si se trata de un *request*), un *status code* (si se trata de un *response*) y un *header* con entradas campo-valor. En un principio se pensó en que las clases tengan también un *body* asociado, ya que tanto los *requests* como los *responses* podrían ocasionalmente tener un cuerpo. Sin embargo esto no se realizó, y la razón es mencionada luego.

Utilizando la herramienta *jFlex*[3], se programaron los parsers que, en base al `InputStream` de un *socket*, construyan el `HttpRequest` o `HttpResponse` correspondiente. En este punto fue cuando se notó que incluir el cuerpo del mensaje dentro de esas clases no era conveniente, debido a que ello implicaba parsearlo por completo y asociárselo a la clase, y no se tenían en cuenta los siguiente factores:

1. Es conveniente devolverle al cliente la información del cuerpo dividida en partes, para que el navegador pueda construir lo que dicha información represente a medida que le va llegando, y no que tenga la obligación de recibir todo el cuerpo para recién visualizar el contenido.

2. El mensaje puede tener el campo `Transfer-encoding: chunked`, con lo cual, si se parsea todo el contenido del cuerpo y una vez hecho esto se lo envía, se pierde la funcionalidad que un paquete `chunked` ofrece (por ejemplo, la generación de páginas dinámicas).

Por lo tanto, se decidió que los parsers tengan métodos para leer el paquete sin cuerpo, y otros para leer el próximo `chunk` del cuerpo si el contenido es `chunked`, o leer los próximos n bytes del mismo si no lo es.

3.3. Compatibilidad con HTTP 1.0

Si bien el proxy funciona con el protocolo *HTTP* en su versión 1.1, debe ser compatible hacia atrás con la versión 1.0.

En principio no se encontraron grandes dificultades, debido a que se testeaba con sitios que devolvían *responses* versión 1.1. Sin embargo, al probar con el proxy transparente utilizado en el ITBA, se descubrió que el funcionamiento no era del todo correcto.

En primera medida, el proxy transparente utilizado en el ITBA suele cerrar las conexiones sin avisar. Esto generó problemas en el manejador de conexiones, que suponía que si una respuesta no aparecía con el header `Connection: close`, no debía cerrar la conexión ya que ésta se mantendría viva (siguiendo el estándar *HTTP 1.1*).

Otro problema que surgió fue el hecho de que el servidor no manejaba respuestas *chunked*. Se presentó la problemática de que el proxy implementado no funcionaba del todo bien con sitios que devolvían respuestas *chunked*, pero dichas páginas funcionaban sin problema al testear desde el ITBA. En relación a esto, luego de testear y leer el RFC, se descubrió que, ante la ausencia de respuestas *chunked*, se debe leer del socket del servidor *origin* hasta que se cierre la conexión. Como este caso fue contemplado, no

[2]Puede encontrarse en el paquete **org.cssc.prototpe.http**
[3]http://jflex.de/

hubo problemas para que la recepción de estas respuestas sea correcta.

Si bien podrían pensarse como problemas menores, el hecho de que el comportamiento del proxy desarrollado en el ITBA difiera del comportamiento del mismo proxy en otro lugar generó confusión y dificultó la solución de otros problemas, que por estos motivos no se podían encontrar.

3.4. Conexiones persistentes

Una de las funcionalidades requeridas para el proxy fue la de implementar conexiones persistentes; es decir, que no se cree una nueva conexión con el servidor *origin* cada vez que se hace un *request*, sino mantener abiertas las conexiones. De esta manera, ante un *request* a un servidor con una conexión ya abierta, se evita tener que abrir una nueva.

En principio, se mantenía abierta una conexión para cada servidor *origin*. Pero esto generaba un funcionamiento ineficiente, ya que no podían haber dos conexiones al mismo servidor simultáneamente, con lo cual las conexiones persistentes reducían la capacidad del proxy. Por lo tanto, se optó por tener, para un mismo servidor *origin*, n conexiones (donde n es configurable), y tener un máximo de m conexiones abiertas.

La implementación realizada se basa en semáforos[4]. Existe primero un semáforo que se inicializa con m permisos. Al solicitarle una conexión a un servidor al `ServerManager`, se pide un permiso de dicho semáforo; al liberarse una conexión, se libera un permiso del mismo semáforo. De manera similar, para cada servidor, existe un semáforo inicializado con n permisos; cuando se desea obtener una conexión para dicho servidor, se solicita un permiso al semáforo correspondiente a ese servidor, y cuando se libera una conexión a dicho servidor, se libera un permiso.

Para determinar si se debe crear una nueva conexión o no, lo que se hace es guardarse por cada servidor una cola de *sockets* abiertos: una vez adquirido el permiso del semáforo correspondiente al servidor, se fija si esta cola está vacía. Si no lo está, entonces significa que hay un *socket* abierto a ese servidor, listo para ser usado; si está vacía, entonces se tiene permiso para crear una nueva conexión, ya que se pudo entrar al semáforo del servidor. Cuando se libera un *socket*, si no está cerrado, se agrega a la cola de conexiones persistentes para el servidor correspondiente: de esta manera, se reusan los *sockets*.

Como hay un máximo de m conexiones persistentes abiertas al mismo tiempo, independiente de cuál sea su servidor, se implementó, lógicamente, un mecanismo para cerrarlas. En las colas de conexiones de cada servidor aparecen solamente *sockets* que no están en uso. Por lo tanto, cuando se necesita crear un nuevo *socket*, si se excede la capacidad de conexiones persistentes, se busca aquel servidor del que no se hayan pedido conexiones hace más tiempo, y se libera un socket que aparezca en dicha cola (o, en su defecto, se sigue buscando una cola que tenga conexiones abiertas). Esta conexión seguro existe debido a la existencia del semáforo con m permisos: este semáforo se asegura de que, en cualquier momento, haya como máximo m threads hablando con algún servidor. Como el thread actual no ha obtenido aún una conexión, hay en ese momento $m - 1$ threads como máximo hablando con algún servidor; como hay m conexiones ya abiertas, hay al menos una que no está siendo usada.

Paralelamente a los problemas de velocidad, aparecieron también otros, más serios, entre los cuales se pueden mencionar:

[4]Ver la clase **org.cssc.prototpe.PersistentSemaphorizedServerManager**

- El encargado de mantener conexiones persistentes con los servidores, después de un tiempo, dejaba de otorgar conexiones y dejaba a los threads bloqueados. Esto se debía a que no se estaba liberando siempre el socket correspondiente a la conexión con el servidor.
- Las conexiones con los clientes nunca se cerraban, y el proxy simplemente se quedaba leyendo de las mismas. Se debió implementar un timeout en dicha lectura, y hacer que se cierren las conexiones cuando se supere ese timeout.
- Se recibían *responses* sucios cuando se reusaba una conexión a un servidor. Esto ocurría debido a que no se leía correctamente la respuesta anterior, con lo que quedaban bytes pertenecientes a la misma; al intentar leer la nueva *response* del siguiente *request*, se leían dichos bytes, generando un error.

En síntesis, la implementación de conexiones persistentes tanto entre el cliente y el proxy como entre el proxy y los servidores *origin* fue la principal causa de problemas en el desarrollo del trabajo práctico, y su solución nunca fue simple, debido a que por la naturaleza del servidor y la existencia de threads concurrentes, resultaba complicado reproducir los bugs.

3.5. Monitoreo

El diseño implementado no contempló, desde un principio, encapsular a los *sockets* en clases específicas del proxy. Las conexiones, así como la entrada/salida, se manejan directamente utilizando la clase `Socket`. Debido a esto, se presentaron importantes problemas al intentar contabilizar la cantidad de bytes transferidos. La solución fue hacer subclases de `InputStream` y `OutputStream` que permitan contabilizar la cantidad de bytes escritos o leídos.

Se presentó además un pequeño problema, relacionado con el monitoreo. Cuando se intentaban contabilizar la cantidad de bloqueos, a veces se contabilizaba más de uno cuando mediante un navegador se accedía a una página bloqueada. Esto era debido a que el navegador intentaba, al mismo tiempo que accedía a la página, buscar el *favicon* de la misma. Inclusive, el navegador reintentaba varias veces ir a buscar el *favicon*. Debido a esto, se concluyó que en realidad la implementación era correcta.

3.6. Configuración remota

La configuración remota de este proxy HTTP se montó sobre el mismo protocolo HTTP ya que ofrecía ciertas facilidades. Por lo tanto es un servicio Web al que se accede mediante otro puerto, también configurable en el XML.

Debido a las características de la implementación, hay ciertos parámetros que no son configurables remotamente y sólo pueden ser cambiados directamente desde el XML de configuración y reiniciando el proxy. Entre ellos, por ejemplo, se encuentra la cantidad de threads que hay en el *pool*, y la cantidad máxima de conexiones persistentes.

Como un usuario se necesita autenticar para poder cambiar la configuración, se decidió que no se puedan modificar los usuarios remotamente, a fines de no perder la autenticación. Se utilizó *HTTP Basic Authentication* para autenticar usuarios, siguiendo los lineamientos que expone la RFC 2617.

El Web Server implementado es iterativo, es decir, sólo atiende de a un HTTP Request. Se decidió hacerlo de esta manera ya que no se supone que sea un servicio con alta concurrencia, sino que simplemente es

accedido por el administrador del proxy para modificar las configuraciones del mismo cuando sea necesario.

Para que el Web Server tenga cierta extensibilidad, se implementó una versión muy reducida de la API de Servlets de Java, haciendo que el comportamiento sea familiar para quien haya usado esa API y quiera implementar algún nuevo servicio en este Web Server. El mapeo de URLs también es realmente muy simple, sólo basta agregar a un mapa el *path* requerido como clave y la clase del Servlet como valor.

El Web Server soporta subir archivos al servidor, en este caso se implementó esto para poder subir nuevos archivos de configuración que reemplacen la configuración actual. Sólo se modificará la configuración de los filtros, el resto del archivo es ignorado.

Este Web Server resultó práctico para exponer también el monitoreo del Proxy HTTP, por lo tanto se creó un servlet para el monitoreo también protegido mediante autenticación, que expone dicha información.

3.7. Servidores y clientes que no "hablan" correctamente el protocolo

Se presentaron dificultades al dialogar con servidores o clientes que no hablan correctamente el protocolo. Por ejemplo, al intentar ver una imagen determinada, se presentó el problema de que no se podía leer la imagen para cargarla en memoria y luego rotarla; esto ocurría porque a pesar de que el *content-type* declarado en la respuesta era *image/jpeg*, el servidor en realidad estaba devolviendo un XML. Esto sucedía en el caso de un request AJAX de una imagen, que al no estar autenticado en twitter.com fallaba y la forma de informar el error era mediante un XML, pero no enviaba el content type correcto (informaba siempre *image/jpeg*).

Otro problema consistió en que algunos navegadores como *Mozilla Firefox* envían *headers* no descriptos por el RFC 2616, como *Proxy-Connection* en vez de *Connection*. Se debió lidiar con estas inconsistencias en el proxy.

3.8. Proxy transparente

El proxy transparente fue probado en un ambiente Linux, usando *firewalls*; más precisamente, el comando `iptables`.

En un principio, se observó que el proxy, utilizado en forma transparente, no tenía el mismo comportamiento que siendo configurado en forma manual en los navgeadores. Sin embargo, luego se aumentó el número de *threads* disponibles en el *pool* y se pudo ver que el funcionamiento era mucho mejor.

Finalmente, se concluyó que, al no ser configurado manualmente, el navegador lanza la misma cantidad de conexiones que cuando no tiene un proxy como intermediario; esto no sucede cuando se le especifica un proxy, ya que realiza un número limitado de conexiones. Al iniciar muchas conexiones, esta cantidad eventualmente puede superar a la que hay disponible en el *pool* de *threads*, provocando que algunos *requests* deban esperar para hacerse y recibir finalmente un *response*.

Luego de esto, se observó que el proxy en modo transparente funcionaba correctamente.

4. Limitaciones

El proxy desarrollado tiene las siguientes limitaciones:

- No soporta *pipelining* de requests al servidor. Por ello, si bien se reusan las conexiones a los servidores cuando llegan distintos requests, la eficiencia de ello deja que desear.
- No soporta todos los métodos *HTTP*, solo `GET`, `HEAD` y `POST`; con lo cual hay ciertas páginas que no funcionan del todo bien porque usan otros métodos.
- Para realizar filtrado de páginas por *Content-length*, es necesario parsear el contenido completo de un paquete. Por lo tanto, no se le puede enviar la información de a pedazos al cliente en el caso de que no haya un header *content-length*: se debe leer todo el contenido y luego enviar la información, con lo cual se pierde en eficiencia.

5. Posibles extensiones

En esta sección se mencionan algunas posibles extensiones que podrían realizársele a esta implementación del proxy *HTTP*.

- Podría agregarse la funcionalidad de ser usado como servidor caché, opción muy utilizada en varios servidores proxy.
- Sería útil almacenar estadísticas sobre qué sitios son visitados por ciertas direcciones IP destino, y en base a ello modificar las respuestas de dichos sitios con información destinada a algún fin en partícular (por ejemplo, publicidad). Si se implementara la funcionalidad de servidor caché, también sería posible, en base a la estadística, determinar que páginas podrían tener más prioridad de almacenamiento en el servidor que otras. Las conexiones persistentes también podrían mantenerse por un tiempo determinado por dichas probabilidades.
- Podrían agregarse prioridades a las IPs destino, con el fin de posibilitarle el uso de un mayor ancho de banda a unas u otras, basándose en dicha posibilidad. También podrían utilizarse esas medidas para permitir mayor o menor cantidad de conexiones simultáneas a servidores *origin* para ciertos clientes.
- Además de los bloqueos ya existentes, sería posible bloquear el acceso a una URI o IP luego de transferida cierta cantidad de bytes entre un cliente y un servidor *origin*.

6. Conclusión

El desarrollo de un proxy *HTTP* fue aportador desde el punto de vista de la comprensión del funcionamiento del protocolo.

Fue muy interesante el hecho de tener la posibilidad de interceptar la información enviada por un cliente o un servidor, y poder modificarla a gusto. Se observó que esto puede tener muchas utilidades, ya sea las implementadas o las mencionadas en las posibles extensiones. Es impactante ver como, configurando un proxy, se puede interceptar la información a, o desde, por ejemplo, un determinado rango de IPs, para de esta forma "robar" datos privados que no estuvieran encriptados, o utilizarlos con fines publicitarios. Todo esto (mediante un proxy transparente) se estaría haciendo de forma que el cliente ni siquiera se entere de que sus datos estan a disponibilidad absoluta del que ha configurado el proxy.

Las principales dificultades encontradas tuvieron que ver con que no todas las aplicaciones que lo aplican cumplen el estándar existente. Como en el caso del proxy se tuvo contacto con navegadores, servidores *origin* y otros servidores proxy, se observaron varias prácticas que iban en contra de lo dicho en el RFC. Esto es entendible teniendo en cuenta que el protocolo es muy extenso, y respetar la documentación puede traer algunas deficiencias en cuanto a *performance*; no obstante, se concluye que la implementación de aplicaciones que utilicen el protocolo *HTTP* sería más sencilla si los estándares fueran respetados.

7. Ejemplos de testeo

Se realizaron pruebas de carga para testear el funcionamiento del proxy utilizando la herramienta *Apache JMeter*[5]. Todos los testeos se realizaron utilizando 10 threads, y con el proxy ejecutándose en la misma computadora que los testeos.

En primera medida, se compararon las tres implementaciones que se realizaron de conexiones persistentes de proxy a *origin server* probando pedidos GET a www.google.com.ar. Configurando al proxy correctamente para que permita suficientes conexiones persistentes al servidor, se obtuvo que la implementación basada en semáforos (`SemaphorizedPersistentServerManager`), que poseía múltiples conexiones persistentes por servidor, obtenía un rendimiento de requests respondidos por segundo de alrededor del 15 % por sobre la implementación que no persistía conexiones (`SimpleServerManager`); mientras que ambas fueron claramente superiores a la implementación que solo poseía una conexión persistente por servidor (`PersistentServerManager`), como puede verse en la **Tabla 1**.

Implementación	Throughput (requests por seg.)
PersistentSemaphorizedServerManager	39,6
SimpleServerManager	34,2
PersistentServerManager	4,2

Tabla 1. *Throughput* de requests a http://www.google.com.ar/ por segundo, con 10 threads, para las distintas implementaciones.

Se sospechó que el overhead de las conexiones persistentes jugaría un papel más significativo si el costo de la conexión fuera más pesado: para ello, se realizó un segundo testeo utilizando el método HEAD a http://www.konami.jp/. El tiempo de latencia promedio para este servidor fue de 352 ms, mientras que para http://www.google.com.ar/ era de apenas 13 ms. En la **Tabla 2** se pueden ver los resultados de esta prueba, que corroboraron las sospechas planteadas. En particular, cabe resaltar que, aunque la relación entre la implementación que permite múltiples conexiones persistentes a un mismo servidor y la implementación que permite una única conexión persistente permaneció constante entre ambas pruebas, la relación entre el primero y el que crea una nueva conexión cada vez no se mantuvo; lo que muestra claramente que la performance del SimpleServerManager se degradó al incrementarse el tiempo de latencia.

Implementación	Throughput (requests por seg.)
PersistentSemaphorizedServerManager	21,5
SimpleServerManager	12,6
PersistentServerManager	2,3

Tabla 2. *Throughput* de requests a http://www.konami.jp/ por segundo, con 10 threads, para las distintas implementaciones.

Se concluye, entonces, que la implementación de conexiones persistentes en un cliente *HTTP 1.1* es especialmente importante cuando se habla con servidores con los cuales el tiempo de latencia es alto.

[5] http://jakarta.apache.org/jmeter/

Por otra parte, se probó el funcionamiendo del servidor proxy cuando la cantidad de conexiones de clientes superaba la cantidad de threads. Los resultados no fueron alentadores: en la **Tabla 3** se pueden ver las performances que lograron los distintos `ServerManager` con la misma prueba de pedidos GET a http://www.google.com.ar/ al bajar la cantidad de threads en el pool a 5 (manteniendo la cantidad de threads utilizado para la prueba en 10). Básicamente, la performance bajó a la mitad. Por lo tanto, es crucial que el numero de threads en el pool sea lo suficientemente grande como para poder contener a todos los clientes que fueran a utilizar el proxy.

Implementación	Throughput (requests por seg.)
PersistentSemaphorizedServerManager	20,5
SimpleServerManager	18,5
PersistentServerManager	4,4

Tabla 3. *Throughput* de requests a http://www.google.com.ar/ por segundo, con 10 threads, para las distintas implementaciones, usando un pool de 5 threads para atender a los clientes.

Basados en pruebas en los navegadores más populares, se determinó que la cantidad de threads sugerida para que funcione como proxy (en particular en el caso de que se desempeñe como proxy transparente) debía ser de al menos 70 threads por cada cliente en potencia. En general, si funciona como proxy configurado, es suficiente una cantidad mucho menor de threads por cliente en potencia, como ser 10. Esto se debe a que los navegadores, al saber que existe un proxy, limitan la cantidad de conexiones a un número razonable, debido a que interpretan que superar ese número no brindaría mejoras en performance; en cambio, si no conocen de la existencia del proxy, generan una nueva conexión por cada servidor distinto al que accedan, con lo cual la cantidad de conexiones es mucho mayor.

Se sugiere, entonces, por ejemplo, que si el servidor proxy funcionara en un laboratorio de 20 computadoras y el mismo estuviera configurado en los navegadores, que se utilicen 200 threads para el pool; o bien 1400, si se desea que actúe como proxy transparente. Es comprensible que esto demande muchos recursos: en ese caso, se sugiere bajar el timeout de las conexiones persistentes a los clientes y la cantidad de threads.

8. Guía de instalación

A continuación se detalla una guía de instalación para sistemas *Linux y Mac OS X*.

1. En el directorio raíz, se encuentra un archivo `JAR` con el proyecto en su última revisión. En caso en que quiera utilizar este `JAR`, diríjase directamente al paso 3. Si desea recompilarlo, diríjase al siguiente paso.

2. Ubicado en el directorio raíz, ejecute
   `$> compile.sh`
   Esto generará un nuevo `JAR`, sobreescribiendo al ya existente. Si se ejecutó con éxito, podrá observar la leyenda
   `SUCCESS. Please execute run.sh`

3. Ejecute
   `$> run.sh`
   Observará la leyenda
   `Starting CSSC HTTP Proxy...`
   que indica que el proxy está siendo iniciado. Luego de esto, el proxy estará funcionando en los puertos indicados luego de dicha leyenda.

A continuación se detalla una guía de instalación para sistemas *Windows*.

1. En el directorio raíz, se encuentra un archivo `JAR` con el proyecto en su última revisión. En caso en que quiera utilizar este `JAR`, diríjase directamente al paso 3. Si desea recompilarlo, diríjase al siguiente paso.

2. Ubicado en el directorio raíz, ejecute
   `$> compile.bat`
   Esto generará un nuevo `JAR`, sobreescribiendo al ya existente.

3. Ejecute
   `$> run.bat`
   El proxy está siendo iniciado. Luego de esto, el proxy estará funcionando en los puertos indicados.

9. Guía de configuración

El proxy es muy flexible en cuanto a configuración. La misma se realiza mediante un archivo `config.xml`, ubicado en el directorio en el cual se esté ejecutando el proxy.

Aquí se explica qué permite configurar cada tag válido en el XML:

- `<proxyPort>`: Puerto en el cual el Proxy escucha conexiones.
- `<remoteServicesPort>`: Puerto en el cual el Proxy corre el servicio Web para configuraciones remotas.
- `<threadPoolSize>`: El tamaño del *pool* de *threads*.
- `<maxPersistantServerConnections>`: Cantidad máxima de conexiones persistentes.
- `<maxPersistantServerConnectionsPerServer>`: Cantidad máxima de conexiones persistentes por origin server.
- `<loggingFileName>`: Nombre del archivo en donde el Proxy hará *logging* de los *requests* y *responses* que vaya procesando.
- `<clientKeepAliveTimeout>`: Timeout en las conexiones hacia clientes.
- `<serverConnectionPersistentTimeout>`: Timeout en las conexiones persistentes hacia servidores.
- `<filters>`: Especifica un conjunto de filtros. Se aplican en el orden explicitados.
- `<filter>`: Define un filtro. Un filtro debe tener `<conditions>` y `<actions>`.
- `<conditions>`: Condiciones para aplicar un filtro. Pueden ser `<OS>`, `<browser>` y/o `<origin-IPs>`.
- `<OS>`: Especifica para qué Sistema Operativo es aplicado el filtro.
- `<browser>`: Especifica para qué navegador web es aplicado el filtro.
- `<origin-IPs>`: Define un conjunto de IPs para las cuales el filtro es aplicado.
- `<IP>`: Especifica una dirección IP (En el caso de `<origin-IPs>` no soporta expresiones regulares).
- `<actions>`: Especifica un conjunto de acciones a realizar si se cumplen las condiciones anteriormente especificadas.
- `<block-all-accesses>`: Se debe especificar un valor `TRUE` o `FALSE`, que determinará si los usuarios que cumplen las condiciones de este filtro pueden acceder a algún contenido o a ninguno (se les bloquea todo acceso).
- `<blocked-IPs>`: Especifica un conjunto de IPs a las cuales no se podrá acceder si se cumplen las condiciones de este filtro. Es decir, si se coloca la IP de www.itba.edu.ar, no se podrá acceder a ese dominio mientras mantenga esa IP bloqueada.
- `<IP>`: Especifica una dirección IP (En el caso de `<blocked-IPs>` soporta expresiones regulares: “24.232.32.*” es válido como regla).

- `<blocked-URIs>`: Especifica un conjunto de URIs a las cuales no se podrá acceder si se cumplen las condiciones de este filtro. Soporta expresiones regulares de estilo "http://www.google.com/*".
- `<URI>`: Especifica una URI en particular perteneciente al tag `<blocked-URIs>`.
- `<blocked-MediaTypes>`: Especifica un conjunto de *Media Types* que son bloqueados a los usuarios que cumplen las condiciones de este filtro.
- `<MediaType>`: Especifica un MediaType para el tag `<blocked-MediaTypes>`.
- `<max-content-length>`: Especifica el máximo *content-length* que se les permitirá descargar a los usuarios que cumplan las condiciones de este filtro.
- `<transform>`: Especifica un conjunto de transformaciones aplicables a los responses de los requests hechos por los clientes.
- `<l33t>`: Transformará la response (si la misma es *text/plain*) a formato l33t.
- `<images180>`: Transformará las imágenes aplicándoles una rotación de 180 grados.
- `<admin-users>`: Conjunto de usuarios administradores que tienen autorización para acceder a los servicios de configuración y monitoreo.
- `<user>`: Define un usuario dentro de `<admin-users>`.
- `<name>`: Define el nombre de usuario de un `<user>`.
- `<pass>`: Define la contraseña de un `<user>`.

10. Ejemplos de configuración

Aquí se puede ver un archivo de ejemplo que es válido para correr el Proxy:

```
<?xml version="1.0" encoding="UTF-8"?>

<config>
        <proxyPort>8080</proxyPort>
        <remoteServicesPort>8082</remoteServicesPort>
        <threadPoolSize>50</threadPoolSize>
        <maxPersistantServerConnections>30</maxPersistantServerConnections>
        <maxPersistantServerConnectionsPerServer>
                20
        </maxPersistantServerConnectionsPerServer>
        <loggingFileName>log.txt</loggingFileName>
        <clientKeepAliveTimeout>30000</clientKeepAliveTimeout>
        <serverConnectionPersistentTimeout>30000</serverConnectionPersistentTimeout>

        <filters>
                <filter>
                        <conditions>
                                <OS>Linux</OS>
                        </conditions>
                        <actions>
                                <block-all-accesses>TRUE</block-all-accesses>
                        </actions>
                </filter>
                <filter>
                        <conditions>
                                <origin-IPs>
                                        <IP>192.168.1.104</IP>
                                </origin-IPs>
                        </conditions>
                        <actions>
                                <transform>
                                        <l33t>FALSE</l33t>
                                        <images180>TRUE</images180>
                                </transform>
                                <max-content-length>1000000</max-content-length>
                        </actions>
                </filter>
                <filter>
                        <conditions>
                                <browser>Chrome</browser>
                        </conditions>
                        <actions>
                                <blocked-IPs>
                                        <IP>193.145.222.100</IP>
```

```
                                </blocked-IPs>
                                <blocked-URIs>
                                        <URI>http://www.lanacion.com.ar/*</URI>
                                </blocked-URIs>
                                <block-all-accesses>FALSE</block-all-accesses>
                                <transform>
                                        <l33t>FALSE</l33t>
                                        <images180>TRUE</images180>
                                </transform>
                                <blocked-MediaTypes>
                                        <MediaType>text/xml</MediaType>
                                </blocked-MediaTypes>
                        </actions>
                </filter>
        </filters>
        <admin-users>
                <user>
                        <name>ssamra</name>
                        <pass>1234</pass>
                </user>
                <user>
                        <name>jscaruli</name>
                        <pass>passfour</pass>
                </user>
        </admin-users>
</config>
```

Aquí se puede ver un archivo de ejemplo que es válido para ser subido como nuevo archivo de configuración (se removió lo ignorado, pero podría estar sin problemas):

```
<?xml version="1.0" encoding="UTF-8"?>

<config>
        <filters>
                        <filter>
                                <conditions>
                                        <OS>Mac OS</OS>
                                </conditions>
                                <actions>
                                        <blocked-IPs>
                                                <IP>200.200.*.200</IP>
                                        </blocked-IPs>
                                        <transform>
                                                <l33t>TRUE</l33t>
                                                <images180>TRUE</images180>
                                        </transform>
                                </actions>
                        </filter>
```

```
                <filter>
                        <conditions>
                                <browser>Firefox/3</browser>
                        </conditions>
                        <actions>
                                <blocked-IPs>
                                        <IP>192.168.1.1</IP>
                                        <IP>192.168.2.*</IP>
                                        <IP>85.62.96.168</IP>
                                        <IP>193.145.222.100</IP>
                                </blocked-IPs>
                                <blocked-URIs>
                                        <URI>http://www.gaiaonline.com/*</URI>
                                </blocked-URIs>
                                <block-all-accesses>FALSE</block-all-accesses>
                                <transform>
                                        <l33t>FALSE</l33t>
                                        <images180>TRUE</images180>
                                </transform>
                        </actions>
                </filter>
        </filters>

</config>
```

11. Diseño de la aplicación

En esta sección se hace una descripción del diseño de clases de la aplicación.

11.1. Modelado de paquetes *HTTP*

La primer decisión en cuanto a diseño consistió en definir cómo se representarían los paquetes *HTTP*.

Las clases que modelan estos paquetes pueden encontrarse en el paquete `http` y, como se mencionó brevemente en la sección en que se explica el parseo, consisten en una superclase `HttpPacket`. Ésta encapsula el estado común de *requests* y *responses*, que son la versión y el *header* (modelado por la clase `HttpHeader`).

`HttpRequest`, que hereda de `HttpPacket`, representa un *request*. Además del estado obtenido de su clase padre, también contiene el método del request (en este caso, `GET`, `HEAD` o `POST`) representado por la clase `HttpMethod`, y el *path*.

`HttpResponse`, también clase hija de `HttpPacket`, es la encargada de representar un *response*. El estado de esta clase está compuesto por el *status code* (clase `HttpResponseCode`) y la *reason phrase*.

Como se dijo anteriormente, el *body* de un paquete no se encuentra en las clases que modelan los paquetes, pues esto implicaba almacenarla en un arreglo de bytes, con lo que se perdía la funcionalidad provista por el encabezado *chunked*.

11.2. Parseo de paquetes *HTTP*

En el paquete `parsers` puede verse las clases utilizadas para construir `parsers` para los paquetes `HTTP`.

La función de estas clases fue descripta en la sección 3.2. En la raíz del paquete se encuentra la clase `HttpParser`, superclase abstracta, y sus hijas `HttpRequestParser` y `HttpResponseParser`. Dentro del paquete `parsers.lex` pueden verse las clases generadas por *Lex* a partir del código fuente presente en el directorio `src/main/resources/lex`.

11.3. Núcleo de la aplicación

En el paquete `net` se encuentra el núcleo de la aplicación. La misma está básicamente compuesta por:

- Una clase *singleton* `Application`, cuyo método `main` crea la única instancia de la misma clase y la corre.
- Una clase `Logger`, que se encarga de registrar los requests y responses que pasan por el proxy. Sólo existe un objeto de este tipo en la aplicación, y es instanciado por la clase `Application`.
- Una clase `ApplicationConfiguration`, que posee principalmente métodos para obtener parámetros de configuración. Al igual que el `Logger`, sólo existe una instancia de `ApplicationConfiguration`, y la misma es instanciada por `Application`.
- Una clase `MonitoringService`, también instanciada una única vez por `Application`, que contiene las cuentas de la cantidad de bytes transferidos y de los bloqueos realizados, y permite acceder a los mismos en forma sincronizada.

- Una clase `ClientListener`, abstracta, y una clase `HttpProxyClientListener`, concreta, que contienen la lógica de un servidor *TCP*.
- Una serie de interfaces:
  - La interfaz `ConnectionManager`, que representa un manejador de conexiones entrantes de clientes.
  - La interfaz `ClientHandler`, que describe clases que puedan manejar un `Socket` resultante de la aceptación de un cliente. La diferencia con `ConnectionManager` es que un `ClientHandler` es aquel que especifica las acciones a efectuar relacionadas con la lógica de la aplicación, mientras que un `ConnectionManager` se limita a administrar los `Socket`s para proveer concurrencia.
  - La interfaz `ServerManager`, que especifica métodos relacionados con la adquisición de `Socket`s para un determinado servidor.
- Una serie de implementaciones de las interfaces arriba mencionadas: `ThreadPoolConnectionManager`, para la interfaz `ConnectionManager`; `HttpProxyHandler`, para la interfaz `ClientHandler`; y finalmente `PersistentSemaphorizedServerManager`, para la interfaz `ServerManager`.
- Un paquete `filters` que contiene diversas clases relacionadas con la aplicación de filtros y transformaciones tanto sobre los requests como sobre los responses.

11.4. Configuración del proxy

En el paquete `configuration` se encuentra la clase `ConfigurationParser`, encargada de parsear el archivo *XML* de configuración y poder servirle los parámetros a la clase `ApplicationConfiguration`.

11.5. Servidor *HTTP*

El servidor *HTTP*, utilizado para configuración y monitoreo remotos, está modelado por las clase del paquete `httpserver`. Dentro de éste, se encuentra la clase `ApplicationConfigurationServer`, que tiene la lógica del servidor (aceptación de clientes y atención de sus pedidos).

En el paquete `httpserver.servlets` están las clases que modelan los *servlets* descriptos en la sección 3.6, y en el paquete `httpserserver.model` hay clases que corresponden a autenticación *HTTP* y a paquetes adaptados al funcionamiento de los *servlets* (`HttpServletRequest` y `HttpServletResponse`).

11.6. Diagramas de clases UML

Los diagramas **UML** de las clases que componen al proxy se encuentran anexados al final de este informe.

12. Referencias

- Enunciado provisto por la cátedra
- RFC 2616, HTTP, *http://tools.ietf.org/html/rfc2616*
- RFC 2617, HTTP Authentication, *http://tools.ietf.org/html/rfc2617*
- Apache JMeter: *http://jakarta.apache.org/jmeter/*
- Apache Bench: *http://httpd.apache.org/docs/2.0/programs/ab.html*