INSTITUTO TECNOLÓGICO de BUENOS AIRES - ITBA

**SEGUNDO CUATRIMESTRE 2013**

MODO PROTEGIDO CON GRUB

**Responsables Académicos:**

**VALLÉS, Santiago Raúl**

**SOTUYO DODERO, Juan Martín**

**CODAGNONE, Juan Francisco**

**Alumnos:**

Fontanella De Santis, Teresa

**Legajo 52.455**

Pagnoni, Agustín

**Legajo 52.118**

Pomerantz, Alan

**Legajo 51.233**

TRABAJO PRÁCTICO ESPECIAL

**PROTOCOLOS DE COMUNICACIÓN**

**ÍNDICE**

**Página**

|  |  |
| --- | --- |
| *OBJETIVO* | 3 |
| 1. DESCRIPCIÓN DE LAS APLICACIONES DESARROLLADAS | 3 |
| 1. DESCRIPCIÓN Y ANÁLISIS DE LOS RFC UTILIZADOS | 3 |
| 1. DESCRIPCIÓN DEL PROTOCOLO PROPUESTO | 4 |
| 1. DISEÑO DE LA IMPLEMENTACIÓN | 8 |
| 1. PROBLEMAS ENCONTRADOS EN EL DISEÑO Y LA IMPLEMENTACIÓN | 11 |
| 1. LIMITACIONES DE LA APLICACIÓN | 11 |
| 1. POSIBLES EXTENSIONES | 11 |
| 1. APLICACIONES A UTILIZAR EN DESARROLLO Y TESTING | 11 |
| 1. CASOS DE PRUEBA | 11 |
| 1. CONCLUSIONES | 12 |
| 1. FUENTES CONSULTADAS | 12 |
|  |  |

*OBJETIVO*

***El objetivo del presente trabajo consiste en 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.***

1. DESCRIPCIÓN DE LAS APLICACIONES DESARROLLADAS

Las aplicaciones desarrolladas son dos:

1. **El proxy HTTP propiamente dicho:**

Funciona en el puerto 9090 y puede configurarse en los navegadores.

1. **El servidor PDC:**

Funciona en el puerto 9091 y procesa requests del protocolo PDC para la configuración y monitoreo remoto de datos estadísticos.

1. DESCRIPCIÓN Y ANÁLISIS DE LOS RFC UTILIZADOS

**Los estándares RFC que serán tomados en cuenta para este trabajo son**:

1. **RFC 2616 (“Hypertext Transfer Protocol – HTTP/1.1”):**

Este documento define el protocolo HTTP versión 1.1, el cual será utilizado por el proxy para poder hablar tanto con los clientes (User Agents), como con los servidores. Si bien todo el documento es importante (porque definen las distintas funcionalidades que pueden utilizarse, y que, por eso, el programa tiene que saber cómo se aplican), se enfatizará en:

* + Conexiones persistentes (sección 8): Es decir, conexiones que duren más de un request. Se consideraron necesarias para poder mantener la conexión entre el proxy y el servidor, y que éste pueda enviarle las respuestas al primero, que es transparente (el servidor envía las respuestas al cliente sin saber que éstas pasan por un proxy). Además, reusan conexiones TCP, lo cual mejora a la performance y la eficiencia.
  + “Status codes” (sección 10): Para poder reportar los eventuales fallos que se produzcan en las conexiones (el hostname no es válido, por ejemplo).
  + “Media types” (sección 3.7): Para poder detectar los distintos tipos de datos que se transfieren y efectuar la transformación a formato 133t (donde corresponda).
  + Estructura de los mensajes (secciones 4 – 6) y métodos definidos (sección 9): Para poder parsear correctamente los mensajes del cliente para el servidor y viceversa.

1. **RFC 2045 (“Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies”):**

Ya que HTTP puede manejar el protocolo MIME para especificar el formato del cuerpo de un mensaje, el proxy también debe conocerlo (especialmente para la transformación del texto plano).

1. **RFC 3117 (“On the Design of Application Protocols”):**

Si bien no es un estándar, menciona aspectos muy importantes a tener en cuenta a la hora de definir un protocolo. Conceptos como “framing”, ”encoding”, “reporte de errores”, “asyncronism”, “escalabilidad”, “eficiencia”, “simplicidad”, “extensibilidad”, ”principio de la robustez de Postel”, se tendrán en cuenta para el armado del protocolo para los puntos 1.11 y 1.12 del enunciado del Trabajo Práctico Especial.

1. DESCRIPCIÓN DEL PROTOCOLO PROPUESTO

En las secciones 1.11 y 1.12 del enunciado, se pide la definición de un protocolo que permita realizar monitoreo remoto de la actividad del proxy y configurarlo. El mismo se llama PDC v1.0 y se lo describe a continuación:

**Características:**

* Se basa en la arquitectura cliente/servidor.
* Request/response.
* Orientado a texto.
* Estructura general: cada mensaje (sea request o response) consta de una primera línea, una segunda línea, una serie de headers seguida por un caracter de nueva línea (“\n”) y, de ser necesario, datos.
* El cliente se conecta al servidor por medio del puerto 9091 (aunque puede configurarse).

**Request:**

**Sintaxis:**

[OPERATION] [PARAM] PDC/ [VERSION]

\n

[DATA]

\n

OPERATION= (“GET”|”AUTHENTICATE”|”FILTER”)

Si OPERATION=”GET” => PARAM = (“ACCESSES” |”TXBYTES” |”HYSTOGRAM”)

Si OPERATION=”AUTHENTICATE” => PARAM= “[USERNAME]:[PASSWORD]”

Si OPERATION=”FILTER” => PARAM = (“ENABLED”|”DISABLED”)

**Análisis**

La primera línea indica la operación a realizar, un parámetro acorde a la misma, y el protocolo y versión con la que se está hablando (PDC y la versión correspondiente, que en este caso es 1.0).

Se definen las siguientes operaciones:

* **GET**

Se obtiene información del servidor. Requiere autenticación previa. Los posibles valores de PARAM son:

* ACCESSES: Para obtener la cantidad de accesos al servidor.
* TXBYTES: Para obtener la cantidad de bytes transferidos por el servidor.
* HYSTOGRAM: Para obtener una serie de datos con las que se puede construir el histograma de status codes.
* **AUTHENTICATE**

Loggea al usuario. El valor de PARAM consta de dos partes separadas por “:” : el nombre de usuario y la respectiva contraseña.

* **FILTER**

Habilita/deshabilita la configuración de la transformación a formato 133t. Si PARAM=”ENABLED”, lo habilita y si PARAM=”DISABLED” hace lo contrario.

La sección DATA siempre está vacía, pero la incluimos para futuras mejoras del protocolo.

**Response**

**Sintaxis**

PDC/ [VERSION] [STATUS CODE] [TEXT CODE]

\n

[DATA]

\n

**Análisis**

La primera línea indica el protocolo y versión con la que se está hablando (que serán los mismos que en el request), y el status code y su correspondiente texto (text code) de la operación realizada.

A continuación, se definen los status code con su respectivo texto (su convención está basada en la usada en HTTP):

* Exitoso 2.x.x

Los status code que comiencen con “2” indican que la operación se realizó con éxito. Sus posibles variantes son:

* 200 OK: Se utiliza cuando se envían datos (operación GET).
* 204 NO CONTENT: Se utiliza cuando DATA está vacío (operaciones ADD y DEL).
* Errores del cliente 4.x.x

Los status code que empiecen con un “4” señalan que la operación no pudo realizarse porque el request está mal formado. Corresponde al cliente arreglar esos errores. En la sección DATA, deben tener un texto explicativo sobre el error.

* 400 BAD REQUEST: Error de sintaxis en el request.
* 401 UNAUTHORIZED: La operación solicitada requiere autenticarse y falta el header “Authentication”.
* 404 NOT FOUND: PARAM no es válido para la OPERATION dada.
* 408 REQUEST TIMEOUT: El cliente no realizó un request en el tiempo en que el servidor lo estaba esperando.
* 420 CORRUPTED DATA: Los datos enviados están corruptos o no son los adecuados para el tipo de operación que se quiere realizar.
* Errores del servidor 5.x.x

Los status code que comiencen con un “5” hacen referencia a errores que se produjeron en el servidor y que evitaron que la operación pudiera llevarse a cabo. En la sección DATA, deben tener un texto explicativo sobre el error.

* 500 INTERNAL SERVER ERROR: Se produjo un error no esperado en el servidor que impidió ejecutar la operación.
* 501 NOT IMPLEMENTED: La operación solicitada no se implementó aún para ningún tipo de PARAM.
* 503 SERVICE UNAVAILABLE: El servidor no está disponible. Se usa para cuando se realiza mantenimiento o hay una sobrecarga de requests.

El contenido de la sección DATA, depende de la operación definida en el request:

* OPERATION= “GET”

Depende del tipo de información que se esté pidiendo (dada por PARAM en el request). Si PARAM=”HYSTOGRAM”, los datos serán pares clave-valor (uno por línea), de la siguiente manera:

[HTTP\_STATUS\_CODE] : [NÚMERO]

donde HTTP\_STATUS\_CODE se refiere a un status code de HTTP (y no del protocolo PDC); y NÚMERO, a la cantidad de responses que retornaron dicho status code.

En cualquier otro caso, los datos contendrán una línea con el número en cuestión (sea para cantidad de accesos o bytes transferidos).

* OPERATION= ”FILTER”

La sección DATA debe ser de esta manera: “Transformations [PARAM]”, donde PARAM hace referencia al valor enviado en el request.

* OPERATION= ”AUTHENTICATE”

En caso de que el usuario y la contraseña no fueran válidos, debe contener “Wrong password”.

**Ejemplo 1:**

(habiendo autenticación previa)

**Request**

GET HYSTOGRAM PDC/ 1.0

\n

\n

**Response**

PDC/ 1.0 200 OK

\n

200 : 100

404 : 1000

\n

**Ejemplo 2:**

**Request**

FILTER ENABLED PDC/ 1.0

\n

\n

**Response**

PDC/ 1.0 401 UNAUTHORIZED

\n

Not logged in!

\n

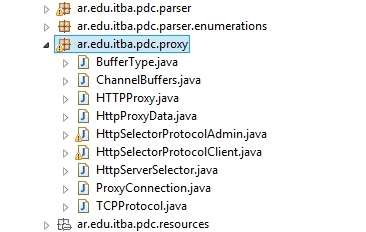
1. DISEÑO DE LA IMPLEMENTACIÓN

La arquitectura del trabajo se basa en el uso de sockets NIO (no bloqueantes), tanto para el proxy como para el servidor PDC.

Teniendo esto en cuenta, a continuación explicamos cada uno de los paquetes:

1. **Paquete “proxy”**

Contiene las clases que usan los sockets NIO.



“HTTPProxy” es la clase que es necesario ejecutar para poner en funcionamiento tanto el proxy HTTP como el servidor PDC. “HttpServletSelector” tiene un método llamado run, por medio del cual utiliza los “HttpSelectorProtocol” (tanto Admin como Client). Estas dos clases implementan una interfaz llamada TCPProtocol, que define métodos para manejar un accept, o cuando se quiere leer o escribir en el socket.

En el “HttpSelectorProtocolClient” se define el selector para un cliente HTTP que utilice al proxy. Tiene un mapa de SocketChannel y ProxyConnection, en donde guarda cada conexión establecida.

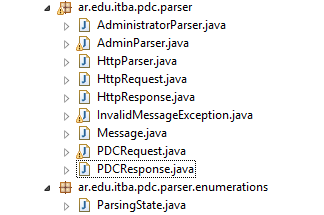
“ProxyConnection” modela una conexión del cliente con el proxy, y del proxy con el servidor.

En el “HttpSelectorProtocolAdmin” se define el selector para un cliente PDC. De manejo parecido al “HttpSelectorProtocolClient”, tiene un mapa de SocketChannel y ChannelBuffers, en donde guarda cada conexión establecida.

“ChannelBuffers” modela una conexión del cliente con el servidor PDC, y contiene un buffer de escritura, otro de lectura y un PDCRequest en donde se va almacenando de a poco el request para poder procesarlo.

1. **Paquete “Parser”**

Contiene las clases utilizadas para el parseo y construcción de paquetes HTTP y PDC.



“Message” es una clase abstracta que modela el comportamiento básico de cualquiera de los mensajes (sean HTTP o PDC).

“HttpParser” parsea un mensaje HTTP haciendo uso de una máquina de estados, y lo guarda en un HttpRequest/Response según corresponda.

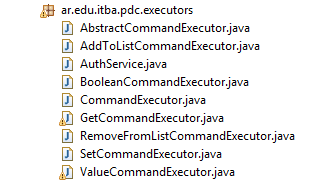
“AdminParser” implementa la interfaz “AdministratorParser”, y parsea un request PDC y guarda esa información en un PDCRequest. Una vez terminado de parsear, se analiza el tipo de Request y, en base a eso, se retorna un PDCResponse con la respuesta. La lógica de qué comando ejecutar se encuentra en PDCRequest.

1. **Paquete “Filters”**

Contiene a “StatisticsFilter”, que se encarga de acumular y los datos necesarios para las estadísticas; y a “TransformationFilter”, que se encarga de realizar la tranformación de text/plain al formato 133t.

1. **Paquete “Executors”**

Contiene aquellas clases encargadas de efectuar los distintos comandos.



AuthService permite autenticarse, GetCommandExecutor permite obtener valores de los Filter (Statistics y Transformation, mencionado anteriormente).

1. **Paquete “Configurations”**

Contiene aquellas clases encargadas de manejar las propiedades de configuración. “PortConfiguration” obtiene los datos de los puertos en donde escuchan tanto el proxy como el servidor PDC y el usuario y contraseña para loggearse, almacenados en el archivo “portConfiguration.properties” del mismo paquete. “ConfigurationCommands” obtiene los datos de si están habilitados o no las transformaciones, etc, del archivo “parsedCommands.properties”, del paquete resources.

1. **Paquete “Logger”**

Contiene una única clase llamada “HTTPProxyLogger”, encargada de escribir los logs en el archivo “logs/log.out” (ubicado dentro del proyecto). Para esto se utilizó la librería “sfl4” (autorizada por la cátedra), debido a su practicidad.

1. PROBLEMAS ENCONTRADOS EN EL DISEÑO Y LA IMPLEMENTACIÓN

A la hora del diseño e implementación de la aplicación, nos encontramos con varios problemas:

1. El tipo de sockets a usar: NIO o bloqueantes. Aun cuando los primeros son más difíciles de implementar que los segundos, elegimos los sockets NIO debido a que utilizan de manera más eficiente los recursos, y tienen buena performance cuando hay una gran cantidad de clientes de forma simultánea.
2. Con respecto a la implementación de los parsers, primero habíamos hecho parsers orientados a línea de texto (debido a que, tanto HTTP como PDC son de esa manera) pero, dado que las conexiones que se establecen son TCP (y éste es un protocolo orientado a bytes), decidimos cambiar dichos parsers por otros orientados a bytes.
3. LIMITACIONES DE LA APLICACIÓN

La principal limitación encontrada en la aplicación es el manejo de los paquetes “chunked”, es decir, aquellos en donde la información viene fragmentada. Esto también implica que no se puede realizar la transformación al formato 133t de los mensajes con Content-Type “text/plain”.

Otra limitación se basa en que el throughput es bastante modesto.

1. POSIBLES EXTENSIONES

Como toda elaboración humana, nuestra aplicación puede extenderse, especialmente arreglando las limitaciones mencionadas en el punto anterior.

1. CASOS DE PRUEBA
2. Usando el navegador Google Chrome y habiendo configurado al proxy, ir a la página www.albahaka.com.ar.

Se intentará realizar una simulación de N conexiones simultáneas, con el objetivo de medir la capacidad de manipulación de conexiones que tenga el proxy, dada la arquitectura seleccionada.

Probar que efectivamente el proxy, en caso de no tener ningún filtro, deje pasar requests del tipo GET, POST y HEAD. En caso contrario que envíe el código de error correspondiente. Se probará enviar varios requests con distinta frecuencia al servidor proxy, por ejemplo un Nginx, que sirva un recurso cuyos valores sean conocidos para probar la correctitud del procesamiento de los request/response que atraviesen al proxy. En el caso de accesos, se medirá en base a la cantidad de sitios accedidos diferentes. La cantidad de bytes transferidos se chequea contra los datos del recurso servido por el servidor proxy, contemplando la cantidad de veces que se accedió. Utilizando dos computadoras que lo corran, dada la configuración en la red para que el flujo de request/response atraviese ambos proxies, se harán las mismas pruebas que para un proxy, haciendo prueba del funcionamiento del encadenamiento de proxies.

1. CONCLUSIONES

Llegando al final de este trabajo, podemos decir que logramos el objetivo propuesto: armar un proxy HTTP. Pero también desarrollamos capacidades como programar con sockets, armar un protocolo, evaluar las limitaciones de nuestro trabajo (cosa que nunca resulta fácil), y toparse con las discrepancias que puede haber entre lo determinado por los estándares RFC y su implementación en la práctica.

1. FUENTES CONSULTADAS

* ***Apuntes de la Cátedra.***
* ***Consultas a la Cátedra****.*