Universidad Autónoma de Madrid

Escuela politécnica superior

Trabajo de fin de grado

EVALUACIÓN DE PRESTACIONES DE TRÁFICO WEB

Grado en Ingeniería Informática

Carlos Gonzalo Vega Moreno
Tutor: Javier Aracil Rico

Mayo 2013

EVALUACIÓN DE PRESTACIONES DE TRÁFICO WEB

AUTOR: Carlos Gonzalo Vega Moreno

TUTOR: Javier Aracil Rico

HPCN Research Group
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo 2013

Resumen

Resumen

Este trabajo de fin de grado sobre evaluación de prestaciones de tráfico web abarca tanto la creación de una herramienta de procesado del tráfico web (Disector de ahora en adelante) mediante la disección de paquetes HTTP, así como, un ejemplo de posible representación de los datos obtenidos en la práctica real mediante al disector utilizando una interfaz web .

La principal motivación para el desarrollo de este disector es tener una herramienta con la que analizar los retardos en las respuestas a peticiones HTTP en una red concreta, de forma online u offline mediante ficheros de tráfico, y así poder estudiar el comportamiento de la red analizada y realizar ajustes que permitan mejorar las prestaciones del tráfico web.

Este trabajo forma parte de un proyecto real con el grupo de investigación High Performance Computing and Networking de la EPS UAM y satisface las necesidades de un problema real en una serie de redes de Sudamérica. Actualmente el proyecto se está probando allí con datos de esas redes.

Esta herramienta se diferencia de otras ya existentes por las prestaciones que ofrece tanto en velocidad del análisis como en la gestión eficiente de recursos que hace que su ejecución en entornos de altas exigencias satisfaga las necesidades de tiempo real de estos. Es una herramienta pensada para ser usada en redes con grandes flujos de datos y miles de conexiones simultaneas por segundo. Además proporciona opciones de filtrado y procesamiento en paralelo que permiten focalizar mejor el análisis a realizar permitiendo ahorrar tiempo de procesado.

El trabajo también analiza un sistema concreto de representación de los datos obtenidos mediante mapas y gráficos de altas prestaciones que permitan dibujar un gran número de puntos sin que afecte al rendimiento de la interfaz. El mapa representará la red concreta sobre la que se han obtenido los datos y los distintos enlaces analizados.

Palabras Clave

Disector de tráfico, HTTP, Análisis, Redes, Alto Rendimiento, Interfaz Web, Gráficos, Mapa

Summary

Summary

This degree project about the performance evaluation of the web traffic covers both the creation of a web traffic processing tool (From now on, "Dissector") by dissecting HTTP packets, as well as an example of data representation of the dissector's obtained outcomes on a web interface.

The main motivation of this dissector's development is being able to analize the delays between responses and their HTTP requests in a particular network, online or offline by using traffic files, and consecuently evaluate the behaviour of the analized network and make fixes to improve the web traffic performance.

This work is part of a real project with the High Performance Computing and Networking research group of the EPS UAM and satisfies the needs of a real problem in a number of networks in South America. Currently the project is being tested there using data from those networks.

This tool differs from the current ones in the offered performance both in processing speed, as well as resources management which allows its execution on high performance environments with real time needs. This tool is intended to be used on networks with huge amounts of traffic and thousands of concurrent connexions per second.

This degree project also assesses an specific representation of the gotten data by using maps and high performance graphs that are able to draw a huge number of points without an interface performance loss. The map represents the particular network from which the data have been obtained and the analized links.

Keywords

Traffic Dissector, HTTP, Analysis, Networking, High Performance, Web Interface, Graphs, Map

Agradecimientos

En primer lugar a mi tutor Javier Aracil y al equipo del laboratorio HPCN por darme la oportunidad de trabajar con ellos. También quiero agradecer a todos los que han colaborado en la realización de las pruebas y resolución de los problemas surgidos.

Dado que este es el trabajo final de la carrera quería agradecer a todos los que me han ayudado a realizarla ya sea directa o indirectamente. A los compañeros por siempre estar ahí cuando surgían problemas o dudas, a los amigos, a ti Marina y a mi familia por el apoyo, paciencia y ánimos dados durante todo este tiempo.

Por último quiero felicitar a los delegados de esta promoción y las asociaciones de alumnos por habernos ayudado en los temas academico administrativos de la carrera, desde proporcionar información sobre las optativas y el trabajo final de grado hasta los trámites de quejas y reclamaciones, gracias por vuestro trabajo, hará mejorar el sistema.

Índice general

Índice de figuras XIII

Glosario de acrónimos XV

1. Introducción 1

1.1. Motivación del trabajo . . . 1

1.2. Objetivos y enfoque . . . 1

1.2.1. Objetivos del disector . . . 2

1.2.2. Objetivos de la representación de datos . . . 2

1.3. Metodología y plan de trabajo . . . 2

1.4. Contenido del documento . . . 2

2. Análisis y solución del problema 5

2.1. Análisis de requisitos . . . 5

2.1.1. Requisitos del disector de tráfico . . . 5

2.1.2. Requisitos de la interfaz web . . . 6

2.2. Análisis y solución del problema . . . 6

2.2.1. Análisis y solución del problema del disector de tráfico . . . 6

2.2.2. Análisis y solución del sistema de representación de datos . . . 8

3. Tecnologías a utilizar 11

3.1. Lenguaje de programación . . . 11

3.2. Librerías externas . . . 11

3.2.1. Librerías externas utilizadas por el disector . . . 12

3.2.2. Librerías externas utilizadas por la interfaz web . . . 13

3.3. Intermediario entre el disector y la interfaz web . . . 13

4. Diseño 15

4.1. Estructura . . . 15

4.1.1. Estructura del disector 15
4.1.2. Estructura de la interfaz web 17
4.2. Implementación 18
4.2.1. Implementación del disector 18
4.2.2. Implementación de la interfaz web 21

5. Desarrollo **23**
5.1. Introducción 23
5.2. Ciclo de vida 23
5.3. Cambios en el diseño 24
5.3.1. Cambios en el diseño del disector 24
5.3.2. Cambios en el diseño de la interfaz web 26
5.4. Inserción de nuevos requisitos 27

6. Pruebas y resultados **29**
6.1. Método de pruebas 29
6.2. Experimentos del disector de tráfico 29
6.2.1. Libnids vs No Libnids 29
6.2.2. Aprovechamiento de los sistemas RAID 31
6.2.3. Retardo frente a URL o dominio 31
6.2.4. Portabilidad 32
6.3. Experimentos de la interfaz web 32
6.3.1. Flot vs Google Chart Tools 32
6.3.2. Portabilidad 32
6.4. Experimentos del sistema completo 33
6.5. Conclusiones 34

7. Conclusiones **35**

8. Mejoras futuras **37**
8.1. Mejoras en el disector 37
8.1.1. Imprimir resumen por URL o dominio 37
8.1.2. Realizar pruebas con redes de 10 y 40Gbps 37
8.1.3. Uso de las cabeceras HTTP 37
8.2. Mejoras en la interfaz web 38
8.2.1. No utilizar Flash 38

8.2.2. Uso de gráficos Retardo/Dominio 38
8.2.3. Gráficos en tiempo real 38
8.2.4. Topografía de la red en los nodos de la interfaz web 38

Bibliografía y Referencias **41**

Índice de figuras

2.1. Ilustración de transacción . . . 7
2.2. Un grupo de usuarios mandando peticiones con distintas conexiones . . . 7
2.3. Clasificación de conexiones, peticiones y respuestas . . . 8
2.4. Boceto de la interfaz de representación de datos . . . 9
2.5. Retardo medio en segundos para cada URL . . . 10

4.1. Ilustración de los dos enfoques o versiones de la aplicación . . . 16
4.2. Ilustración del establecimiento de conexión . . . 16
4.3. Estructura hashvalue . . . 19
4.4. Estructura packet_info . . . 19
4.5. Liberación de nodos . . . 20
4.6. Paquete HTTP . . . 21
4.7. Ejemplo de index.html . . . 22

5.1. Ilustración del desarrollo iterativo . . . 24
5.2. Google Chart Tools Annotated Time Line . . . 26

6.1. Gráfico comparativo del uso de memoria entre ambas versiones en escala *log* 30
6.2. Retardo medio en segundos para cada URL . . . 31
6.3. Interfaz web finalizada . . . 33

8.1. Ejemplo de topología de red . . . 39

Glosario

- **Libpcap**: Librería que permite tratar archivos PCAP y capturar paquetes de la interfaz de red.
- **PCAP**: Packet Capture. Paquete con información del paquete capturado o leído con libpcap
- **Glib**: Librería de propósito general que permite el uso de otros tipos de datos no estándar del lenguaje C así como tablas Hash, Árboles y un mejor tratamiento de hilos y semáforos, entre otros.
- **BPF**: Berkeley Packet Filter. Permite filtrar paquetes para hacer más eficiente el análisis de estos.
- **RRDtool**: Round Robin Database Tool.
- **Handshake**: Procedimiento por el cual se establece una conexión entre un cliente y un servidor.
- **Callback**: Función que es llamada al producirse un evento. En este proyecto es llamada cuando un nuevo paquete es analizado. Se encarga de procesar ese nuevo paquete.
- **Pipelining**: Método de realizar peticiones por el cual se realizan varias peticiones seguidas en grupo y se espera a las respuesta de todas antes de realizar más peticiones. La alternativa es realizar una petición y esperar hasta que llegue la respuesta para realizar una nueva petición.

1
Introducción

1.1. Motivación del trabajo

La principal motivación del proyecto es ofrecer una nueva herramienta para el análisis del tráfico web en redes de altas prestaciones. Esta herramienta debe satisfacer unos requisitos de velocidad y uso de recursos muy específicos para que pueda trabajar con las grandes cantidades de datos generadas por este tipo de infraestructuras.

La causa primera motivante del proyecto es la necesidad real de evaluar las prestaciones ofrecidas en una red de Sudamérica como parte de un proyecto con el laboratorio High Performance Computing and Networking de la Escuela Politécnica Superior en la Universidad Autónoma de Madrid.

Esta necesidad real también motivó la búsqueda de una forma de representación de los datos mediante el dibujo de los nodos y enlaces de la red sobre un mapa y la asociación de gráficos desplegables a dichos enlaces para facilitar el acceso y visualización de los datos.

1.2. Objetivos y enfoque

Los objetivos y enfoque se dividen en dos partes claramente diferenciadas: la herramienta de análisis (Disector) y la herramienta de representación de los datos obtenidos.

En general los objetivos son analizar los distintos diseños y métodos para implementar una herramienta que cumpla los requisitos (explicados en el próximo capítulo) que suponga una diferencia en cuanto a las herramientas disponibles actualmente y diseñar un método de representación de los datos para el problema real explicado anteriormente.

1.2.1. Objetivos del disector

- Diseñar una herramienta que permita un análisis eficiente, rápido y eficaz del tráfico en una red de altas prestaciones y alto flujo de datos.
- Presentar una solución al problema que mejore las herramientas actuales y ofrezca nuevas prestaciones.
- Desarrollar la herramienta con un ciclo de vida iterativo que conlleve la realización de varios prototipos y cambios en el diseño durante el proceso de desarrollo y hagan que la aplicación final resultante este pulida y perfeccionada. Los detalles sobre el ciclo de vida del proyecto se explican con mayor detalle en el capítulo de desarrollo.
- Realizar pruebas que confirmen lo anteriormente citado. También durante las distintas fases del desarrollo iterativo.

1.2.2. Objetivos de la representación de datos

- Diseñar un sistema de representación que permita una visualización eficiente de los datos y esté preparada para la carga de datos que va a representar.
- Presentar una solución al problema que cumpla con los requisitos (especificados en el capítulo siguiente) establecidos.
- Desarrollar la aplicación con un ciclo de vida de nuevo iterativo que conlleve la realización de varios prototipos que serán mejorados entre sus distintas versiones.
- Realizar pruebas que confirmen lo anteriormente citado.

1.3. Metodología y plan de trabajo

La realización del trabajo ha consistido en una primera parte de análisis de requisitos (los cuales son explicados en el capítulo siguiente anterior) donde se han tenido en cuenta los distintos aspectos y necesidades a cumplir para después efectuar un diseño consecuente a ello.

Se ha realizado un análisis de las distintas opciones para satisfacer los requisitos así como un análisis de prestaciones entre ellas.

El desarrollo iterativo ha permitido ir perfeccionando la aplicación y corrigiendo sus defectos hasta conseguir cumplir con los requisitos establecidos. Se han ido añadiendo nuevos requisitos en cuanto a que surgían nuevas necesidades y problemas en el proyecto real.

1.4. Contenido del documento

El documento ofrece una introducción y motivación del proyecto realizado, explica los problemas a resolver, establece unos objetivos claros y unos requisitos a cumplir para solucionar el problema propuesto.

En los aparatados intermedios se expone el diseño y desarrollo de la solución al problema con los distintos cambios, mejoras y correcciones realizadas durante el desarrollo del proyecto.

Al final del documento se explican las pruebas realizadas, tanto los escenarios de pruebas como los resultados así como las conclusiones obtenidas de las pruebas.

Finalmente un último capítulo que expone mejoras futuras del proyecto.

2

Análisis y solución del problema

2.1. Análisis de requisitos

A continuación se explican tanto los requisitos funcionales como los no funcionales referentes al disector de tráfico y a la interfaz web.

2.1.1. Requisitos del disector de tráfico

Requisitos funcionales del disector de tráfico

El requisito más importante es mostrar por pantalla los tiempos de retardo entre la petición y la respuesta de una conexión HTTP ordenados en función de la hora de llegada de la respuesta y los datos de la conexión como las IPs y puertos de origen y destino, la marca temporal, la URL y el código y mensaje de respuesta.

Estos datos los deberá mostrar a partir de un fichero o una lista de ellos en formato PCAP, RAW o directamente en tiempo real utilizando la interfaz indicada en los parámetros de ejecución del programa. El procesamiento de la lista de ficheros será o bien secuencial o en paralelo, según se indique al ejecutar el programa.

Otros requisitos funcionales son la posibilidad de introducir como parámetro un filtro BPF (BSD Packet Filter) y/o de URL para tener la posibilidad de filtrar el tráfico analizado por cualquiera de sus parámetros (IP de origen, destino, puerto, etc), como los paquetes HTTP analizado por su URL respectivamente. También debe permitir mostrar la salida del programa en distintos formatos con distintos fines.

Por último deberá mostrar una barra de progreso para el archivo (o archivos) que está procesando. Esta barra deberá mostrar también la velocidad media a la que se están leyendo los datos.

Requisitos no funcionales del disector de tráfico

El rendimiento tanto en velocidad como en gestión de los recursos de memoria es primordial para poder analizar grandes cantidades de datos usando el menor número de recursos y aprovechando la velocidad tanto de los discos que contienen los datos como de la interfaz si el análisis es en modo online. Esto permitirá analizar el tráfico más o tan rápido de lo que se genera.

Otro factor importante es la portabilidad ya que su ejecución debe ser posible en distintos sistemas Linux/Unix con el menor número de dependencias.

2.1.2. Requisitos de la interfaz web

La interfaz debe ser capaz de dibujar un gran número de puntos en los gráficos y comportarse con fluidez a pesar de haber varios gráficos.

Requisitos funcionales de la interfaz web

Los datos se mostrarán en una interfaz agradable y lo más sencilla posible para facilitar el entendimiento y la consulta de los datos. Deberá permitir hacer zoom y navegar por el gráfico en el eje temporal. Al pasar el ratón sobre los distintos puntos se mostrarán los datos correspondientes al punto seleccionado, tanto la marca temporal como el retardo calculado.

Requisitos no funcionales de la interfaz web

La interfaz web deberá ser capaz de dibujar un gran número de puntos en los gráficos y comportarse con fluidez a pesar de haber varios gráficos.

2.2. Análisis y solución del problema

En esta sección se analizan tanto los problemas como las soluciones a implementar de las distintas partes del proyecto. Algunos detalles que se explican en esta sección serán determinantes en los capítulos de diseño y desarrollo.

2.2.1. Análisis y solución del problema del disector de tráfico

Como ya se explicó en la sección de requisitos el principal objetivo del disector de tráfico es mostrar la diferencia entre el tiempo de una petición y su respuesta.

Cada petición tiene un origen, un destino, una marca temporal y la URL que estamos pidiendo. Como es obvio la respuesta también tiene un origen, un destino y una marca temporal, además del dato correspondiente a la URL.

Todas las peticiones y respuestas con un mismo origen y destino van a través una misma conexión. Y todas las peticiones son resueltas en orden, es decir, si se realizan

varias peticiones seguidas en una misma conexión las respuestas llegarán en el mismo orden en que se realizaron las peticiones. Siendo, por tanto, imposible que se obtuviese en primer lugar la respuesta a la segunda petición que a la primera.

La siguiente figura ilustra una transacción entre un usuario y un servidor.

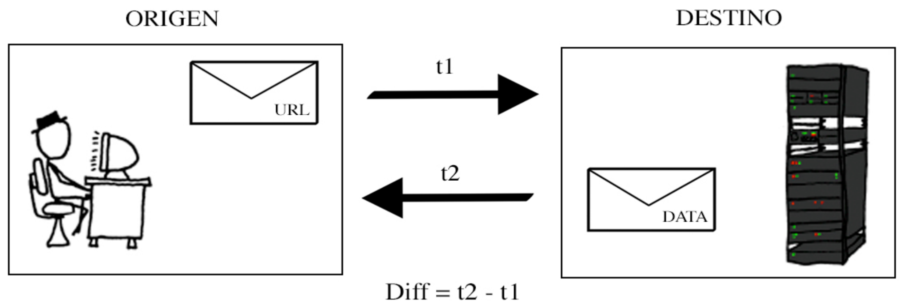

Figura 2.1: Ilustración de transacción

En un caso real cada usuario realizará un sinfín de peticiones, en distintas conexiones, es decir, con distintos destinos. En la figura cada sobre ilustra una petición y cada color una conexión con un destino diferente. Además de una gran cantidad de peticiones realizadas por un usuario tenemos un gran número de usuarios, es decir, de orígenes.

En consecuencia el número de peticiones y respuestas se vuelve enorme.

Figura 2.2: Un grupo de usuarios mandando peticiones con distintas conexiones

La tarea más importante es organizar estas peticiones y respuestas por su origen y destino de tal forma que tengamos una lista de conexiones (Origen, Destino) y a cada una de esas conexiones otra lista asociada de pares (Petición, Respuesta) a los cuales llamaremos transacciones.

Las peticiones pueden ser realizadas de una en una esperando la respuesta antes de hacer otra petición o en ***pipeline*** y esperar todas las respuestas en orden.

En el apartado de diseño se explica con más detalle la solución implementada con las estructuras y algoritmos utilizados tanto para clasificar y almacenar las conexiones como para emparejar las respuestas con sus correspondientes peticiones.

En la siguiente figura tenemos tres conexiones (O_1D_1, O_1D_2 y O_2D_1) cada una con una lista de transacciones. En las cuales algunas peticiones han sido satisfechas (P_1R_1, P_2R_2, P'_1, R'_1, etc.) y otras no (P_3 y P''_2).

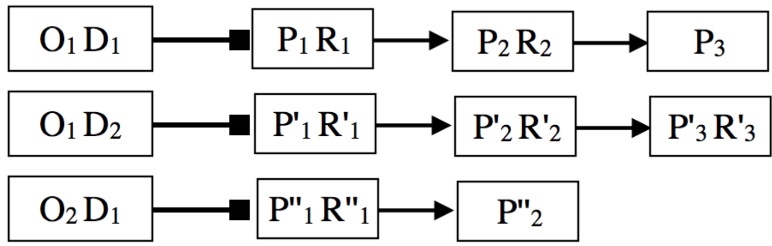

Figura 2.3: Clasificación de conexiones, peticiones y respuestas

Al llegar la respuesta se calculará la diferencia de tiempo entre las marcas temporales de la petición y la respuesta para posteriormente imprimir en pantalla o en fichero los datos de la transacción.

2.2.2. Análisis y solución del sistema de representación de datos

La otra parte del proyecto consiste en un sistema de representación de datos que permita una visualización rápida y sencilla. En este apartado se explicará un boceto de diseño de la interfaz. Los algoritmos y metodologías que hacen que tenga un buen rendimiento se explicarán en los capítulos de diseño y desarrollo.

Mapa con gráficos asociados

La interfaz mostrará un mapa con enlaces entre distintos puntos clave. Cada enlace llevará asociado un gráfico que mostrará la media de las diferencias de tiempo entre las peticiones y respuestas en un determinado momento. Este mapa se mostrará mediante alguna clase de evento asociado a los enlaces (click, por ejemplo)

Dado que para un mismo instante de tiempo habrá varias peticiones de distintas conexiones en el gráfico tan solo se mostrará la media entre todos los datos de un determinado instante. Como puede verse en la siguiente figura, el gráfico permite, mediante el "MiniMap"navegar por los datos del eje X tal como se recoge en los requisitos.

Resolución de los datos

Para que el rendimiento del gráfico no se reduzca por el hecho de dibujar un gran número de puntos la resolución de los datos será mayor para los instantes recientes y menor para los instantes pasados. De esta forma en vez de tener un punto por cada instante (segundo) en los datos de hace más de un mes, tendremos, por ejemplo, la media de cada día. De esta forma el número de puntos a dibujar se reduce considerablemente.

Este tema se aborda con más detalle en el apartado **3.3. Intermediario entre el disector y la interfaz web** del capítulo **3. Tecnologías a utilizar**.

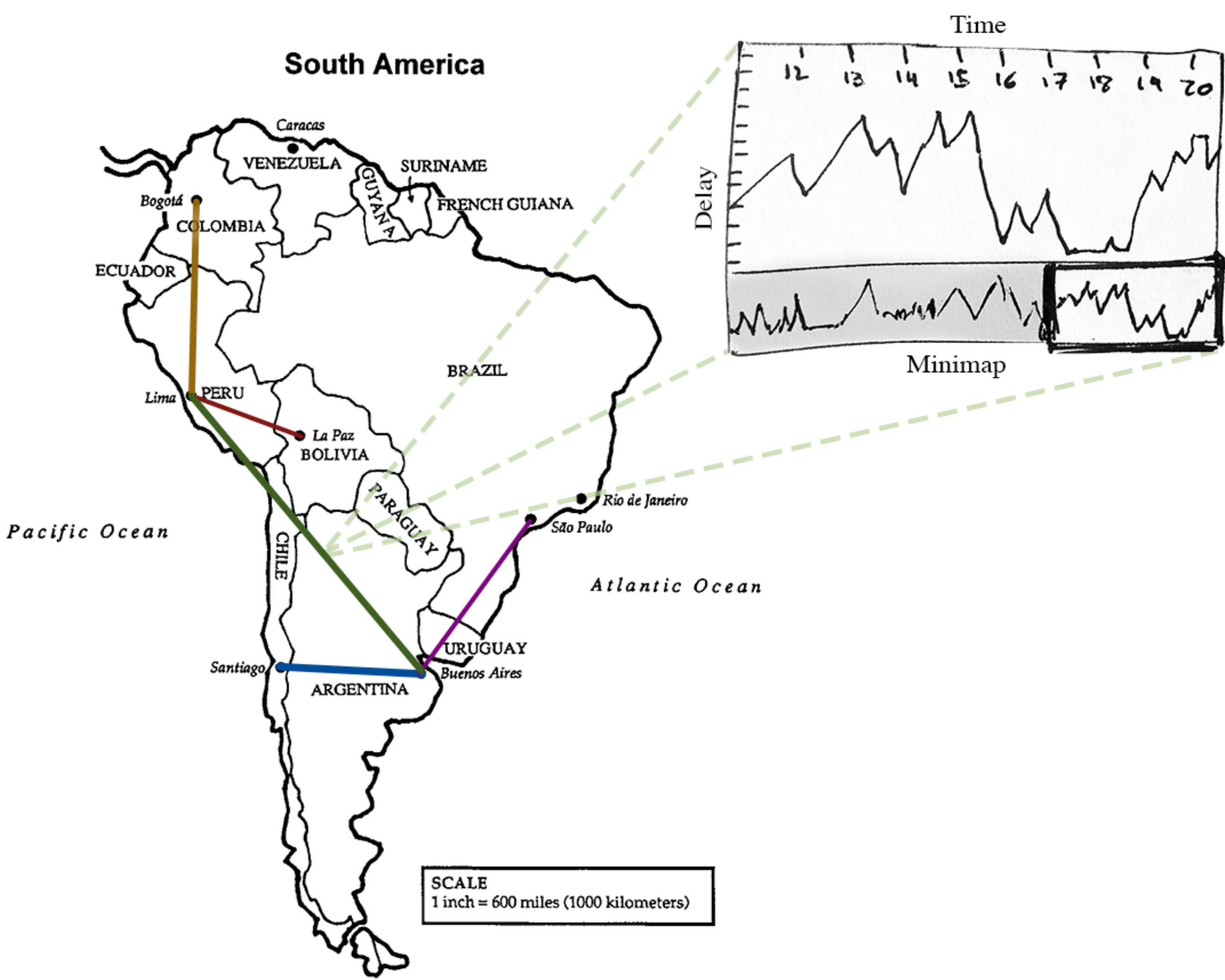

Figura 2.4: Boceto de la interfaz de representación de datos

Gráfico de retardos según la URL

Otro gráfico interesante es el que muestra los retardos de una determinada URL o dominio, (y por tanto, de un determinado servicio o servidor). Estos gráficos muestran los retardos de cada una de las URLs o dominios para que sea posible determinar de un vistazo si algún dominio tiene algún problema en los retardos.

A continuación se muestra un gráfico que representa el retardo medio para cada uno de esos dominios.

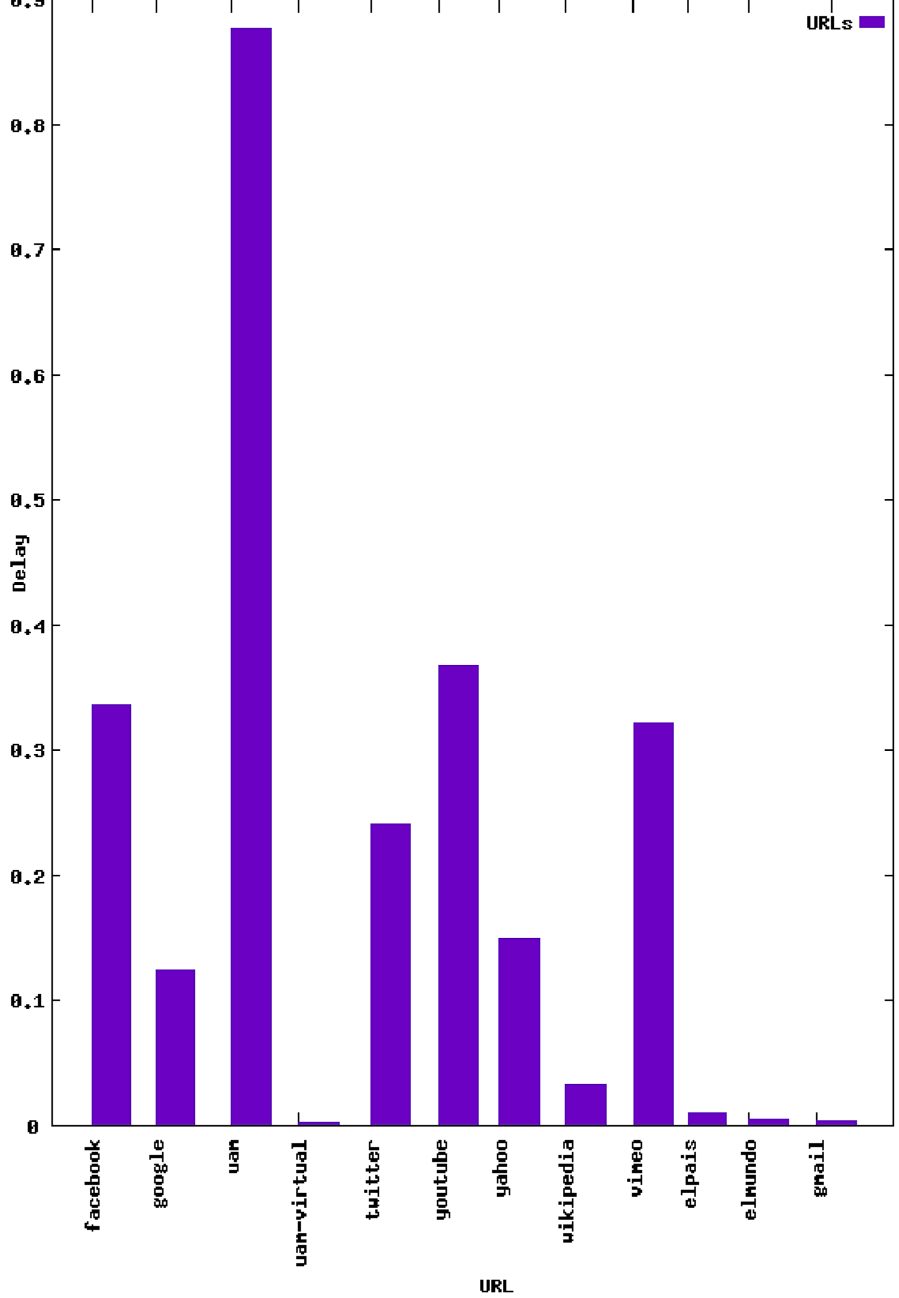

Figura 2.5: Retardo medio en segundos para cada URL

3

Tecnologías a utilizar

3.1. Lenguaje de programación

La aplicación será programada en C ya que ofrece un alto rendimiento ya que permite utilizar características de bajo nivel y poder realizar una implementación óptima. Además es un lenguaje con un gran número de librerías útiles y optimizadas que ayudan a la hora de implementar una aplicación eficiente. A pesar de que cueste más tiempo programar en este lenguaje las ventajas que ofrece frene a los inconvenientes hacen que merezca la pena programar en este lenguaje.

Este lenguaje fue escogido respecto a otros como Python porque además de permitir operaciones a bajo nivel permite controlar mucho mejor el uso de memoria realizado por el programa.

En cuanto a la representación de los datos se ha elegido la implementación de una interfaz web en HTML combinado con el uso de JavaScript y librerías externas para mostrar los mapas y gráficos. Debido a la simplicidad de la interfaz web no es necesario usar ningún tipo de lenguaje orientado al desarrollo web de contenido dinámico como PHP.

3.2. Librerías externas

Esta sección se divide en dos partes las librerías referentes al disector y las utilizadas en la interfaz web. Cabe recordar que la utilización de una librería en el disector no tiene ninguna consecuencia en la elección de librearías en la interfaz web (y viceversa) ya que son módulos totalmente independientes que se unen entre sí mediante un método intermedio que se explicará en apartados posteriores.

3.2.1. Librerías externas utilizadas por el disector

A continuación se exponen las librerías externas utilizadas para el desarrollo del disector. Todas son librerías de código abierto que permiten la libre distribución del código.

Libpcap

La principal librería externa utilizada en el desarrollo del disector es libpcap la cual permite tanto el tratamiento de los archivos PCAP (packet capture) como la captura directa desde la interfaz de red. Es una librería openSource muy potente que permite el procesado de paquetes de red. Será fundamental en el desarrollo de esta aplicación.

Fue desarrollada por los creadores de tcpdump del Lawrence Berkeley National Laboratory, otra gran herramienta ampliamente utilizada en el mundo de las redes.

Una de las cualidades más importantes de esta librería es la posibilidad de aplicar un filtro BPF (Berkeley Packet Filter) que permita no copiar los paquetes que no nos interesan desde la capa de kernel al nivel de usuario y por tanto quita carga a la CPU y reduce el espacio ocupado en el buffer.

Glib

Otra de las librerías externas más importantes usadas en el desarrollo de esta aplicación. Permite tipos de datos que no están disponibles de forma estándar en C como listas enlazadas, tablas hash, árboles, etc. También ofrece un mejor tratamiento de hilos y semáforos que el estándar de C.

En el desarrollo de esta aplicación se ha utilizado para la implementación de la tabla hash que guarda las distintas conexiones; los hilos, entre ellos el recolector de conexiones agotadas; y el uso de semáforos para el acceso a la tabla hash.

Libnids

Es una librería desarrollada por Rafal Wojtczuk que emula el stack de Linux, permite la desfragmentación de paquetes IP y el ensamblado del tráfico TCP lo cual permite un mejor tratamiento de los paquetes HTTP ya que mediante esta librería dispondremos de ellos ya ensamblados y sin estar repartidos en fragmentos IP.

Esta librería supone facilidades en el desarrollo del disector pero también supone unas ciertas desventajas. En capítulos posteriores se realizan pruebas entre dos enfoques distintos, uno que utiliza esta librería y otro que no. En ese capítulo se analizan las ventajas y desventajas. Ese análisis supondrá un punto de inflexión en el desarrollo de la aplicación.

NDleeTrazas

Esta librería, proporcionada por el grupo de investigación de la Escuela Politécnica Superior, High Performance Computing and Performance de la UAM es un envoltorio (wrapper) de las funciones de **libpcap** para poder utilizar tanto archivos en formato RAW como archivos en formato PCAP.

3.2.2. Librerías externas utilizadas por la interfaz web

En este apartado se exponen las librerías referentes al desarrollo de la interfaz web para la representación de los datos obtenidos con el disector. Son todas libres y salvo alguna restricción de uso (número de visitas a la web y/o ánimo de lucro) se pueden utilizar de forma gratuita y sin limitaciones. Todas ellas son librerías de código JavaScript.

API de Google Maps

Es un conjunto de funciones JavaScript que permiten introducir mapas en páginas web. Permite todo tipo de personalización tales como la introducción de líneas entre dos puntos, curvas, marcar puntos de interés e incluso colorear el mapa de forma personalizada. En este trabajo se ha utilizado para representar los nodos y enlaces que representan una red ficticia basada en un problema real.

Google Chart Tools

Es una herramienta potente que dispone de distintos tipos de gráficos para mostrar información. Son gráficos que hacen uso de JavaScript y Flash y permiten dibujar una gran cantidad de puntos sin que ello afecte al rendimiento web. Tiene la desventaja de que es necesario Flash para visualizar los gráficos pero esa tecnología permite una rápida renderización de los gráficos.

Flot

Es una librería JavaScript para trazar gráficos de forma sencilla. No utiliza Flash sino canvas para dibujar los gráficos. Utiliza jQuery.

jQuery

Es una biblioteca de JavaScript que facilita el uso de este lenguaje simplificando y reduciendo el código. Facilita, también, notablemente el uso de AJAX y eventos de JavaScript.

3.3. Intermediario entre el disector y la interfaz web

Para probar el sistema (Disector, web) es necesario un intermediario que guarde los datos obtenidos mediante el disector para después suplir los gráficos con datos. Para ello se ha utilizado una base de datos RRDtool (Round Robin Database Tool).

RRDtool es una herramienta que trabaja con una base de datos con una cantidad de datos fija, definida en el momento de la creación de la base de datos. Esta herramienta es muy popular en el mundo de las redes y es utilizada por otras como ntop. Permite definir distintas resoluciones para los espacios temporales, por ejemplo podemos establecer que se guarde 1 muestra por cada dos segundos (haciendo la media entre los dos segundos) durante un día entero, 1 muestra por minuto para la última semana, 1 muestra por hora para el último mes, etc. De esta forma tenemos más precisión en los datos más recientes y vamos perdiendo precisión según esas muestras se hacen antiguas.

El disector dispone de un método de salida que tan solo imprime la media de retardos que ha habido en cada segundo. Esto facilita la adición de datos a la base RRDtool. Por otro lado un script automático consultará los datos cada cierto tiempo (establecido en función de las necesidades) para generar el archivo que la interfaz web consultará para dibujar los gráficos.

Será necesaria una base de datos por cada gráfico. Normalmente se representará un enlace por gráfico aunque pueden mostrarse varios enlaces en un mismo gráfico para realizar comparaciones.

4

Diseño

4.1. Estructura

En este apartado se explica la estructura de diseño del disector y la interfaz web. Los aspectos relacionados con el rendimiento se explican en el capítulo siguiente (**Desarrollo**) ya que en esta sección solo se consideran los aspectos teóricos a priori del diseño. Los detalles de implementación de ambas partes se tratan en este mismo capítulo en el apartado de **Implementación**.

4.1.1. Estructura del disector

A continuación se van a explicar los dos enfoques que se han tomado para realizar el disector a lo largo del desarrollo de este proyecto y cómo se han diseñado. Se explicarán qué características aportan y qué ventajas y desventajas tienen la una frente a la otra. En el capítulo de desarrollo se explican los motivos por los que finalmente se ha elegido una de ellas y se incluye un análisis de rendimiento entre una versión y otra.

Estructura común entre versiones

Las dos versiones reciben parámetros que configuran características como: la ejecución tanto en modo offline como online, impresión en pantalla o escritura en fichero, salida en un formato u otro, lista de ficheros o ficheros únicos, formato PCAP o RAW, etc.

Después, según la versión del diseño, se establecerán los parámetros para la librería libnids o para la librería libpcap.

En el *callback* ambas versiones realizan la clasificación de las conexiones y el emparejamiento de las respuestas con sus correspondientes peticiones como se explicó en el apartado (**2.2.1 Análisis y solución del problema del disector de tráfico**). Ambos tienen en común la llamada al *callback* pero este es distinto entre las dos versiones.

Finalmente ambas versiones imprimen la información de cada transacción (petición, respuesta) al ser satisfecha la petición en el formato y destino establecidos al inicio.

A continuación se exponen las diferencias entre estas dos versiones. La figura ilustra la estructura de ambas versiones. A pesar de que ambas usan libpcap y un *callback* se ha querido hacer una distinción poniéndolos por separado porque el uso que se les da es distinto.

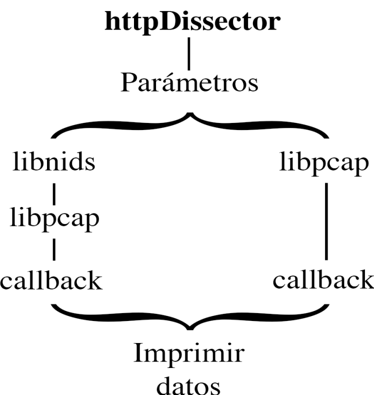

Figura 4.1: Ilustración de los dos enfoques o versiones de la aplicación

Estructura de la versión con libnids

Como se ha explicado en el capítulo anterior (**3. Tecnologías a utilizar**) libnids es una librería que permite el ensamblado de los paquetes IP y el tráfico TCP. De esta forma solucionamos el problema de que los datos de la respuesta (por ejemplo) estén en varios paquetes, lo cual supondría varias llamadas al callback en vez de una, que es lo que tendremos ahora gracias a libnids.

Mediante libpcap lee los paquetes del fichero PCAP para posteriormente realizar el ensamblado de los paquetes TCP/IP. Este ensamblado puede requerir, a priori, bastante tiempo de CPU, este aspecto se estudiará más a fondo en el capítulo de desarrollo.

Libnids nos permite elegir qué conexiones queremos seguir y cuales no, en principio seguiremos todas salvo que por parámetro se indique lo contrario. Por tanto para que libnids nos notifique de la llegada de paquetes debe haber un *handshake* mediante los paquetes SYN, SYN-ACK, ACK como se ilustra en la figura.

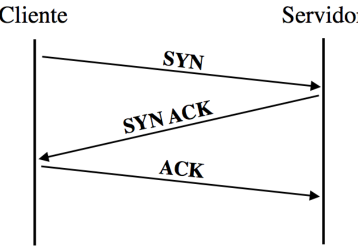

Figura 4.2: Ilustración del establecimiento de conexión

Necesitar un handshake tiene la desventaja de que no podremos tener en cuenta las peticiones y respuestas de una conexión de la cual no tenemos establecimiento de conexión. Por ejemplo las conexiones que se establecieron antes de comenzar la captura de paquetes para crear el fichero PCAP. Si esas conexiones son largas la cantidad de peticiones y respuestas perdidas puede ser considerable.

En el *callback* de esta versión es necesario tener en cuenta los distintos tipos de paquete que libnids nos proporciona, a tener en cuenta:

- **Handshake**: Esta llamada al callback indica que se ha establecido una conexión.
- **Reset**: Indica que hubo un reset de la conexión especificada.
- **Close o Exiting**: Libnids ha terminado de procesar el fichero.
- **Data**: Este paquete lleva datos útiles, *payload*. Es una petición, una respuesta u otra cosa. Estas son las llamadas que nos interesan para clasificar la conexión y la transacción.
- **Fin**: La conexión especificada ha terminado.

Estructura de la versión sin libnids

Como se aprecia en la **figura 4.1** la versión que no hace uso de la librería libnids usa directamente libpcap, sin intermediarios. No ensamblamos los paquetes IP ni el flujo TCP porque solo nos interesan los paquetes que sean peticiones y respuestas HTTP. Esto nos ahorra el tiempo de CPU que usa libnids para realizar el ensamblado. A priori esta versión puede perder información de la URL ya que el paquete que la contiene puede estar segmentado en varios. En el capítulo de desarrollo se trata más a fondo el aspecto técnico de ambas versiones. En cuanto a las respuestas no nos interesa más que la marca de tiempo y con el primer paquete (en caso de segmentación) nos vale.

Por tanto esta versión solo deberá configurar los parámetros de libpcap y procesar los paquetes de forma similar a como se hace en la otra versión. En el apartado de implementación se explica con más detalle el algoritmo de clasificación y emparejamiento.

Ventajas y desventajas a priori de las distintas versiones

La principal ventaja de libnids es que ensambla el tráfico y nos provee de distintas llamadas según el tipo de paquete TCP. Aunque por otro lado puede que ese procesamiento extra corra en nuestra contra ya sea en tiempo o uso de memoria.

Las mayores ventajas de la versión que no hace uso de libnids son el ahorro de tiempo de procesado, no es necesario el establecimiento de la conexión para tener en cuenta las peticiones y respuestas HTTP. Posiblemente use menos memoria.

4.1.2. Estructura de la interfaz web

La interfaz web tiene una estructura mucho más sencilla que la del disector. Se compone de un mapa que abarca toda la ventana del navegador. Sobre ese mapa estarán

dibujados los distintos enlaces entre los nodos (también dibujados). Al hacer click sobre un enlace se mostrará un gráfico en un lado de la ventana que representará los retardos en el tiempo tal y como se explica en el apartado Análisis y solución de la interfaz web.

4.2. Implementación

En esta sección se explican detalles como los algoritmos de clasificación de conexiones, las estructuras con las que se almacenan los datos en memoria, la forma en que se guardan los datos a representar en la interfaz web, etc.

4.2.1. Implementación del disector

El disector tiene distintas estructuras con las que trabaja, las estructuras que se describen a continuación hacen referencia a las de la versión final de la aplicación. En el capítulo de desarrollo se exponen algunos hechos relevantes o cambios que han llevado a la versión actual de las estructuras.

Tabla hash

Para clasificar y almacenar las conexiones utilizaremos una tabla hash donde la clave estará compuesta de: dirección IP de origen (IP_o), puerto de origen (P_o), dirección IP de destino (IP_d) y puerto de destino (P_d). La clave por tanto será la concatenación de estos cuatro datos ($IP_oP_oIP_dP_d$).

Para crear la clave de un paquete tendremos en cuenta si el paquete es una petición o una respuestas para dar la vuelta a los datos en caso de que sea respuesta y que así coincidan las claves con las de las peticiones correspondientes.

El dato asociado a la clave es una estructura que entre otros datos contiene una lista enlazada. De ahora en adelante lo llamaremos **hashvalue**.

Hashvalue

La estructura hashvalue es el nodo principal de una lista enlazada que tiene como nodos la estructura **pair** que está formada por un puntero al siguiente nodo (también de tipo **pair**) y que contiene dos estructuras de tipo **packet_info**, una para la petición y otra para la respuesta.

Hasvalue también almacena el número de peticiones y el de respuestas que hay en la lista enlazada así como cuantos nodos han sido liberados. También tiene una estructura **timespec** que guarda la marca temporal del último paquete añadido a la lista enlazada.

La figura 4.3 muestra el código de la estructura hashvalue.

```
typedef struct _pair {
    struct _pair *next;
    packet_info *request;
    packet_info *response;
}pair;

typedef struct {
    pair *list;
    pair *last;
    int n_request;
    int n_response;
    int deleted_nodes;
    struct timespec last_ts;
} hash_value;
```

Figura 4.3: Estructura hashvalue

Packet_info

Esta estructura contiene todos los datos que componen una petición o respuesta. A destacar: Su URL (en el caso de la petición), el código y el mensaje de respuesta (en el caso de la respuesta), la marca temporal, la IP origen, el puerto origen, la IP destino, el puerto destino, una variable que indica si el paquete es una petición o una respuesta.

En la figura a continuación se muestra el código que define la estructura packet_info.

```
typedef struct _pkt_info{
    struct sniff_ethernet *ethernet; // The ethernet header
    struct sniff_ip *ip;             // The IP header
    struct sniff_tcp *tcp;           // The TCP header
    u_char *payload;                 // Packet payload
    u_int size_ip;
    u_int size_tcp;
    u_int size_payload;
    char ip_addr_src[ADDR_CONST];
    char ip_addr_dst[ADDR_CONST];
    unsigned short port_src;
    unsigned short port_dst;
    short request;
    struct timespec ts;
    char *url;
    short responseCode;
    char response_msg[256];
} packet_info;
```

Figura 4.4: Estructura packet_info

Algoritmo del callback

Cada vez que un paquete nuevo es leído libnids o libpcap llaman al callback. En esta función lo primero que se averigua es si se trata de una petición, una respuesta u otra cosa. Para ello se utiliza la función **parse_packet**.

Lo primero que realiza esa función es una comprobación de que las cabeceras IP y TCP son correctas. A continuación mediante la función **http_parse_packet** del módulo

http.c (que se explicará más adelante) se comprueba si es un paquete HTTP y de ser así si es un GET o un RESPONSE.

Una vez sabemos que es un paquete válido (petición HTTP o RESPONSE) procedemos a insertarlo en la tabla hash. La función **insert_get_hashtable** se encarga de insertar una petición en la tabla hash, para ello comprueba si la clave hash existe, de no ser así se crearía un nuevo **hashvalue**. Si existiera se obtendría el hashvalue asociado. Después, en ambos casos, se añadiría la nueva petición a la lista enlazada y se insertaría el hashvalue en la tabla hash.

En caso de ser una respuesta se llamaría a la función **insert_resp_hashtable** que en primer lugar comprueba que exista la clave en la tabla hash, si no fuera así se devolvería un error. En el caso normal que es que la clave exista se asociará la respuesta a la primera petición de la lista enlazada.

Esto es así debido a que una vez emparejada respuesta con petición el nodo es liberado de la lista enlazada y los datos asociados imprimidos en pantalla o fichero. Por tanto el siguiente nodo pasa a ser el primero y este por supuesto es una petición aún insatisfecha.

La siguiente figura ilustra el procedimiento de liberación de los nodos.

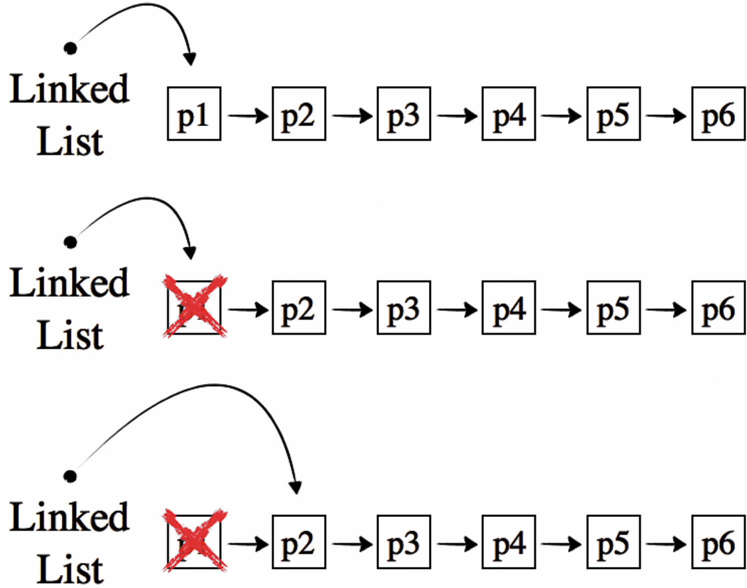

Figura 4.5: Liberación de nodos

Módulo http.c

Este módulo se encarga de procesar los paquetes HTTP, tiene asociada una estructura **http_packet** y una serie de funciones, a destacar:

- **http_get_host**: Devuelve el host del que proviene la petición.
- **http_get_uri**: Devuelve el URI (Identificador uniforme de recursos) de la petición.
- **http_get_response_code**: Devuelve el código de respuesta (por ejemplo, 404).

Este módulo también analiza las cabeceras de las respuestas y crea una lista enlazada con cada una de ellas en un nodo (clave, valor) que guarda el nombre la cabecera (Accept-Encoding por ejemplo) y el valor asociado (gzip, deflate, etc.). También provee una función para buscar sobre esa lista enlazada una determinada cabecera. De momento en este proyecto no se utiliza esta información pero en versiones futuras puede ser útil, véase el apartado **Mejoras futuras** para más información.

Como puede verse en la figura, la estructura **http_packet** almacena el método, versión, URI, host, código y mensaje de respuesta y los datos asociados a la respuesta del paquete HTTP. Además esta implementación hace que el usuario de la librería no tenga acceso a los miembros de la estructura y esta solo pueda ser accesible mediante las funciones de la librería.

```
struct _internal_http_packet{
    http_header *headers;
    char *data;
    http_op op;
    char method[32];
    char version[32];
    char uri[2048];
    char host[256];
    int response_code;
    char response_msg[256];
};
```

Figura 4.6: Paquete HTTP

4.2.2. Implementación de la interfaz web

La interfaz web consiste en una serie de archivos, un ejemplo mínimo tendría lo siguiente:

- **index.html**: Archivo web principal que referenciará a los siguientes.
- **map.js**: Archivo JavaScript que contiene las especificaciones del mapa y las funciones para cargarlo en la web.
- **plotfunctions.js**: Archivo JavaScript que contiene las funciones necesarias para dibujar el gráfico a partir de los datos.
- **muestras.js**: Archivo JavaScript con los datos en formato JSON de un enlace (a dibujar).
- **format.css**: Archivo de estilos CSS.

La siguiente figura muestra un ejemplo de página web (index.html) básico, no se han incluido las URLs de las APIs.

```html
<html>
 <head>
  <link rel="stylesheet" href="format.css" type="text/css">

  <script type="text/javascript" src="...maps_api..."></script>
  <script type='text/javascript' src="...graphs_api..."></script>
  <script type="text/javascript" src="...jquery..."></script>

  <script type='text/javascript' src='plotfunctions.js'></script>
  <script type='text/javascript' src='map.js'></script>
  <script type='text/javascript' src='mxar/muestras.js'></script>



 </head>
 <body onload="initialize_map()">
  <div id="map_canvas"></div>
  <div id="graph_title"></div>
  <div id="hide_graph_button"><img src="minimize.png"/></div>
  <div class="graph" id='BuenosAires' ></div>

  <script>
    $("#hide_graph_button").click(function () {
      $(this).slideUp();
      $(".graph").slideUp();
      $("#graph_title").slideUp();
    });

    plot(mxar_in, mxar_out, 'BuenosAires');

  </script>
 </body>
</html>
```

Figura 4.7: Ejemplo de index.html

5
Desarrollo

5.1. Introducción

En este capítulo se hablará principalmente de los problemas que han ido surgiendo durante el desarrollo del proyecto y de cómo se han tratado. También se hablará a posteriori de aspectos relacionados con el rendimiento de las distintas versiones y algoritmos que no se trataron en el capítulo de Diseño.

También se hablará de cómo se han ido introduciendo nuevos requisitos a lo largo del proyecto.

El próximo apartado habla del ciclo de vida elegido para el desarrollo de este proyecto y los motivos por los que se ha elegido.

5.2. Ciclo de vida

Tanto en el desarrollo del disector de tráfico como en el de la interfaz de representación de datos se utiliza un ciclo de vida iterativo que permite el desarrollo de distintos prototipos o versiones útiles y funcionales.

Se ha elegido este ciclo de vida porque aunque la solución del problema está clara el desarrollo de ella requiere realizar distintas pruebas y probar varias librerías y herramientas para realizarlas hasta encontrar la más adecuada para la solución del problema.

Este ciclo de vida tiene la desventaja de que requiere un cliente muy involucrado en el desarrollo de la aplicación para que dictamine si las necesidades del sistema se están viendo satisfechas. Dado que el rol de cliente en este caso es el propio tutor que realiza las pruebas en el sistema real de México esta desventaja se convierte en una gran ventaja ya que durante el desarrollo del proyecto se ve claramente si se va en el buen camino.

En este proyecto los requisitos iniciales no tienen una gran probabilidad de verse alterados aunque si pueden presentarse nuevos requisitos que conlleven la realización de

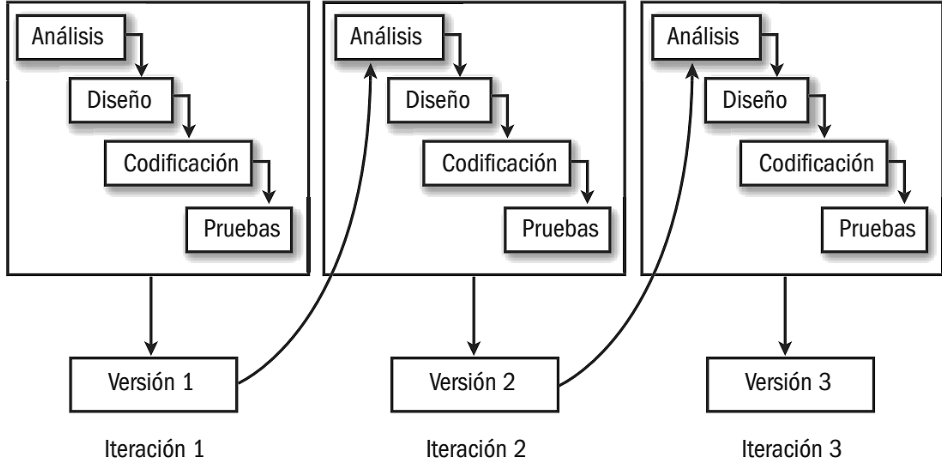

Figura 5.1: Ilustración del desarrollo iterativo

modificaciones en el proyecto. En ningún caso estos nuevos requisitos penalizarán otros, si acaso serán adiciones al anterior prototipo que se verán satisfechas en el ulterior.

Por tanto el desarrollo partirá de una implementación simple de los requerimientos e iterativamente se mejorarán las versiones producidas hasta completar los requerimientos del sistema. En cada iteración se corrigen problemas, se mejoran los módulos, se añaden nuevas funcionalidades o se rediseñan distintas partes del sistema.

5.3. Cambios en el diseño

Durante el desarrollo de este proyecto han aparecido diversos problemas de rendimiento que han requerido la realización de cambios tanto en los algoritmos como en las librerías utilizadas.

5.3.1. Cambios en el diseño del disector

Tabla hash

En el desarrollo del disector uno de los primeros problemas que nos encontramos es que la tabla hash debe ser suficientemente eficiente, estable y rápida como para soportar un gran número de inserciones y eliminaciones.

De esta forma quedó totalmente descartado la implementación de una tabla hash propia salvo que fuese la única opción. Afortunadamente la tabla hash que provee la librería **Glib** resultó más que suficiente para satisfacer las necesidades de alto rendimiento que establecidos en los requisitos del capítulo segundo.

Recolector de basura

Otro de los problemas encontrados rápidamente fue las conexiones perdidas o abandonadas sin cerrar. había peticiones insatisfechas y conexiones que nunca recibían la señal de **FIN** por lo que se quedaban en memoria ocupando recursos.

Para solucionar este problema se diseño un recolector de basura en un hilo a parte que cada 10 segundos despierta y busca entradas en la tabla hash cuya marca temporal del último paquete añadido a la lista enlazada (**last_ts**, ver Sección **packet_info**) haya sido hace más de un minuto.

Para realizar este proceso se ha utilizado la función de la librería **Glib** llamada **g_hash_table_foreach_remove** que tan solo necesita una tabla sobre la que actuar y una función que determine para cada elemento si debe ser eliminado o no.

Este cambio permitió ahorrar una cantidad de memoria que en archivos grandes o en modo online se podría incrementar bastante y causar problemas.

Sistema de liberación de recursos

Inmediatamente después de la creación del recolector de basura el método de liberación de recursos fue mejorado para liberar los nodos de las listas enlazadas y las conexiones tan pronto se pudiera. Al inicio del proyecto se guardaban todos los pares (petición, respuesta) hasta que la conexión finalizara, ya fuese por señal FIN o porque el recolector de basura considerase la conexión cerrada).

La forma en la que se mejoró es la ya ilustrada por la **figura 4.5** en la que se puede apreciar cómo nada más ser satisfecha la petición con su correspondiente respuesta este nodo es liberado de la lista enlazada. De ser el último nodo en la lista enlazada se liberaría toda la conexión eliminando la clave de la tabla hash.

Abandono de Libnids

Tras realizar una serie de pruebas se empezó a notar que libnids no era suficientemente eficiente en velocidad y uso de memoria cuando se utilizaban archivos de tráfico grandes (mayores de 10GB) y dado que este disector debe estar preparado para analizar archivos de incluso varios Terabytes de tamaño se decidió realizar una versión que no usase libnids tal y como se explica en la subsección **Estructura de la versión sin libnids** del capítulo **4. Diseño**.

En la sección **Libnids vs No Libnids** del capítulo de **Pruebas** se explica detalladamente el resultado de la prueba que se realizó.

Las conclusiones fueron que la versión del disector que hacía uso de la librería **libnids** utilizaba tanta memoria que sería impensable utilizarlo con archivos reales. Este uso de memoria es debido al procesamiento interno de los paquetes TCP/IP que realiza la librería.

Además la nueva versión tiene un uso de memoria bastante bajo y constante y aprovecha mejor las altas velocidades de lectura de los sistemas SSD y RAID.

Lista de ficheros

Para la realización de este cambio en el diseño se tuvieron que aislar algunos procedimientos de la función principal (***main***) de tal forma que la función que arranca el sistema y procesa los parámetros introducidos es el proceso padre. Este proceso padre

creará un proceso hijo para que procese el primer archivo. El proceso hijo tiene una copia de todas las variables globales, tabla hash, etc. y procesa el archivo mientras el padre duerme hasta que el hijo termine de procesar el fichero. Una vez termina el proceso hijo este muere y el padre crea otro proceso para el siguiente fichero.

El padre hace de semilla ya que el tiene el estado inicial que utilizarán cada uno de los hijos que vayan a procesar los distintos ficheros.

5.3.2. Cambios en el diseño de la interfaz web

Dado que la complejidad de esta parte del proyecto es bastante más reducida que la del disector no hubo muchos cambios en el diseño durante el desarrollo de la interfaz web.

De Flot a Google Chart

En las primeras fases del desarrollo de esta aplicación se estuvo utilizando una librería sencilla de creación de gráficos mediante JavaScript llamada **Flot**.

A pesar de lo sencillo de su uso no resultaba adecuada y rápida para varios gráficos de numerosos puntos por tanto se buscaron varias alternativas como **Chartools** y **Google Chart Tools**. Entre estas dos ultimas se eligió Google Chart Tools porque entre sus tipos de gráficos tenía el más adecuado a las necesidades del proyecto.

A continuación se muestra un ejemplo.

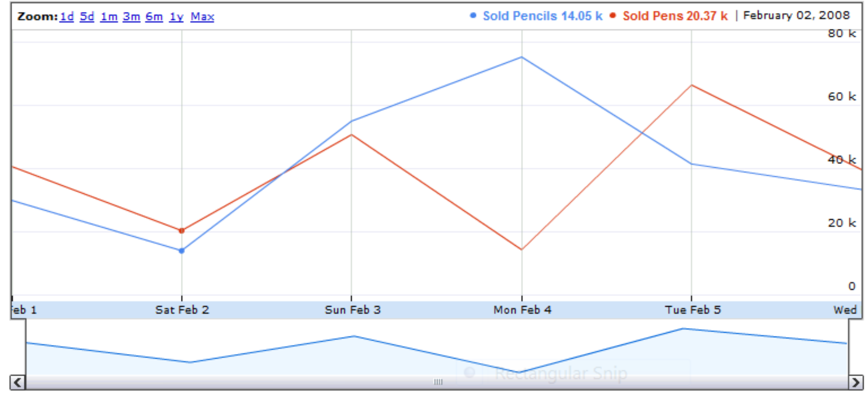

Figura 5.2: Google Chart Tools Annotated Time Line

Para compararla con Flot se realizó una prueba comparativa que consistía en mostrar 10 gráficos con 3.600 puntos cada uno y otra que mostrara 10 gráficos con 36.000 puntos por gráfico. Los detalles de la prueba se pueden ver en el apartado **Flot vs Google Chart Tools** del capítulo de **Pruebas**.

La conclusión breve es que Google Chart Tools ganó con bastante margen a Flot ya que mientras que Flot tarda bastante con la primera prueba de 3.600 puntos Google lo hace mucho más rápido y el tiempo no se incrementa mucho en la prueba de 36.000 puntos mientras que Flot en esta segunda prueba se vuelve insufrible.

5.4. Inserción de nuevos requisitos

Durante el desarrollo del proyecto se han ido añadiendo nuevos requisitos entre cada una de las versiones. Algunos de los requisitos que se añadieron durante el desarrollo del proyecto son:

- **Filtro BPF**: Filtra paquetes para hacer más eficiente el análisis de los datos.
- **Filtro URL**: Filtra las peticiones por URL. Esto permite estudiar los retardos de un dominio o servicio concreto.
- **Formatos de salida**: Se han añadido distintos tipos de formato que permiten distintos usos. Uno de dos líneas por transacción que permite una rápida visualización en pantalla. Otro en una única línea que permite analizarlo rápidamente con erramientas de *scripting* y un tercero que imprime la media de retardos para cada segundo.
- **Lista de ficheros**: Recibe una lista de ficheros que serán procesados uno a uno. Este caso se explica más a fondo en un apartado dedicado (**Lista de ficheros**) en la sección **Cambios en el diseño del disector**.
- **Procesamiento en paralelo**: Permite el procesamiento de varios archivos en paralelo. Un proceso totalmente independiente para cada uno de los ficheros. Todavía está en fase de pruebas.

6

Pruebas y resultados

6.1. Método de pruebas

Las pruebas han sido realizadas en distintos entornos dependiendo de la prueba en sí. Para algunas pruebas sencillas no es necesario un gran ordenador con RAIDs y mucha memoria RAM mientras que otras necesitan todo lo contrario, un sistema puntero de alto rendimiento.

Se añadió una opción de debug (*−−log*) al disector de tráfico que escribe información cada cierto tiempo en un log del sistema sobre el uso de memoria de la aplicación y la velocidad media de lectura desde disco. Con esta información es posible realizar una serie de pruebas.

6.2. Experimentos del disector de tráfico

Los experimentos del disector de tráfico deberán probar que se cumplen los requisitos de velocidad y uso eficiente de recursos que se exigían.

6.2.1. Libnids vs No Libnids

Como se explica en la sección **Abandono de Libnids** del capítulo **5. Desarrollo** se comenzó a apreciar un uso excesivo de los recursos de memoria y una alta necesidad de tiempo para procesar ficheros de tamaño considerable.

Por ello se procedió a la realización de una nueva versión que no hiciera uso de esa librería mediante otro enfoque del problema. Y una vez se tuvieron ambas versiones desarrolladas se realizó una prueba comparativa con el mismo fichero en el mismo entorno para determinar la diferencia de tiempo y uso de memoria.

La prueba se realizó con un ordenador con procesador Core 2 Duo a 2,4 Ghz, disco duro SSD con una velocidad de lectura máxima de 270MB/s y 8GB de RAM. El fichero que se utilizó fue un archivo de tráfico PCAP de 10GB. Este archivo es pequeño si se compara con los ficheros reales de 2TB de tráfico pero la versión de libnids no podía con ficheros más grandes ya que llenaba la memoria RAM.

La velocidad máxima de lectura alcanzada por la versión con libnids fue de 120MB/s con un uso medio de 100MB/s mientras que la nueva versión alcanzó el pico de 270MB/s y tuvo un uso medio de 220MB/s.

Como se puede apreciar en el gráfico, el uso de memoria de la nueva versión (en color verde) es constante y se mantiene en torno a los 6MB mientras que la versión con libnids utilizó 4GB al final de la ejecución. Usó 680 veces más memoria que la nueva versión y tardó un 60

Lo más interesante es que la nueva versión se mantiene bajo y constante en el uso de memoria y aprovecha la velocidad de los discos duros SSD y RAID.

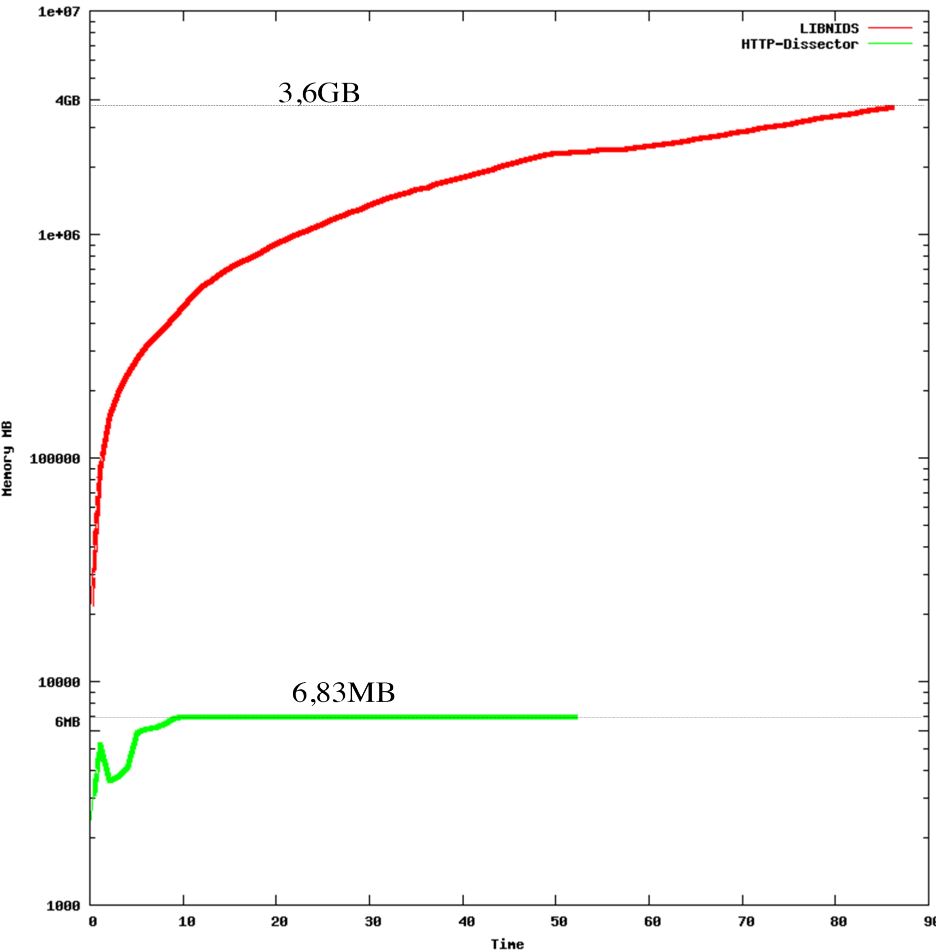

Figura 6.1: Gráfico comparativo del uso de memoria entre ambas versiones en escala *log*

6.2.2. Aprovechamiento de los sistemas RAID

El aprovechamiento de los sistemas RAID de alto rendimiento es crucial para tardar lo menos posible en procesar los datos. La velocidad de lectura de fichero suele ser el único cuello de botella a la hora de conseguir mayor velocidad de procesado ya que el programa y el procesador son lo suficientemente eficientes. Un RAID 0, por ejemplo, permite leer datos simultáneamente de varios discos duros. La velocidad de lectura resultante es igual a la suma de las velocidades de cada disco que compone el RAID.

6.2.3. Retardo frente a URL o dominio

Poder mostrar el nivel de retardo medio para un determinado dominio o URL es realmente útil para apreciar en qué puntos (servidores o servicios) hay un mayor retraso a la hora de satisfacer las peticiones.

La prueba se ha realizado con el mismo archivo y en las mismas condiciones que en el apartado **Libnids vs No Libnids**. Dado que hay un número enorme de URLs distintas se han elegido unas pocas conocidas para mostrar.

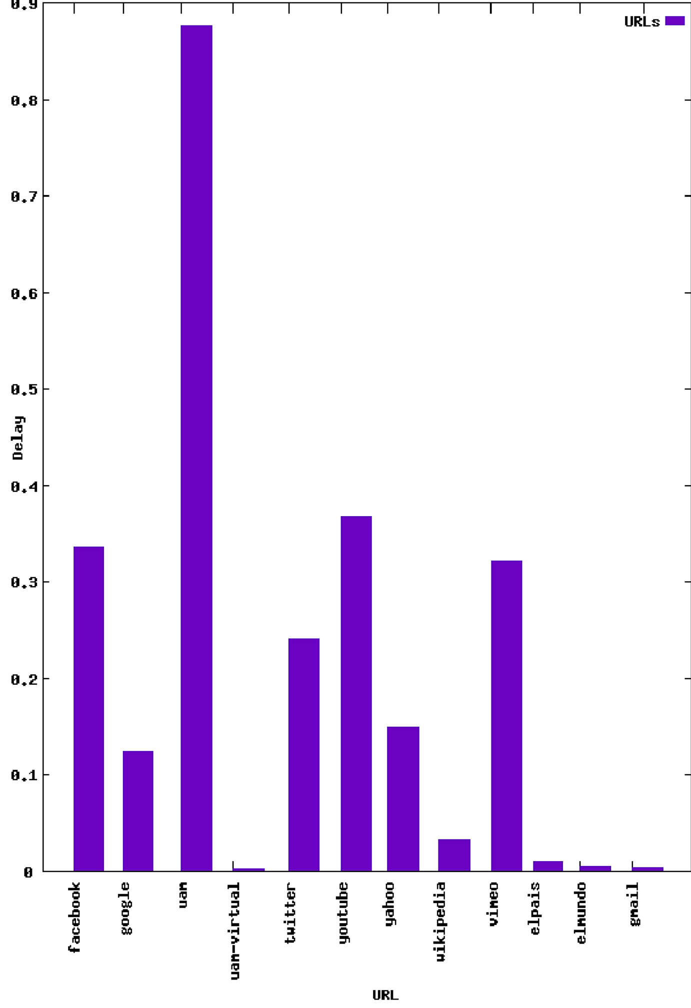

Figura 6.2: Retardo medio en segundos para cada URL

6.2.4. Portabilidad

Se han realizado pruebas de portabilidad en las siguientes plataformas:

- **Mac OS X 18.3 Mountain Lion**: Se ha instalado gtk2 (para glib) y libpcap mediante el gestor de paquetes brew.
- **Ubuntu 12.04**: Se han instalado los paquetes libpcap-dev, libglib-dev.
- **OpenSuse 11.1 SP1**: Se han instalado los paquetes de desarrollo correspondientes de libpcap y glib.
- **Fedora 18**: Se han instalado los paquetes de desarrollo correspondientes de libpcap y glib.

El funcionamiento ha sido exactamente el mismo e incluso soportamos distintas versiones de la librería glib compilando unas funciones u otras en función de la versión instalada de glib.

6.3. Experimentos de la interfaz web

6.3.1. Flot vs Google Chart Tools

Como se explica en la sección **De Flot a Google Chart Tools** hubo ciertos problemas a la hora de representar varios gráficos con una gran cantidad de puntos para cada uno de ellos. Por ello se procedió a la realización de dos pruebas que consistían una en mostrar 10 gráficos con 3.600 puntos y otra en mostrar igualmente 10 gráficos pero con 36.000 puntos.

Dado que Google Chart Tools utiliza tecnología Flash para dibujar los puntos junto con JavaScript para realizar el procesado la velocidad de renderizado fue muy superior a la que ofreció Flot. Este ultimo tardaba bastante en la primera prueba haciéndose insufrible en la segunda. Bloqueaba totalmente el navegador hasta que se representaban todos los gráficos. Google Chart Tools por el contrario no mostraba gran diferencia de tiempos entre la primera y la segunda prueba y por ello fue elegido para representar los datos en la interfaz web.

6.3.2. Portabilidad

La portabilidad de la interfaz web abarca todos aquellos navegadores donde esté instalado Flash Player. En navegadores como Chrome está incluido por defecto. También deben soportar y debe estar activado JavaScript, que hoy en día está en prácticamente la totalidad de los navegadores web.

6.4. Experimentos del sistema completo

Se realizó una prueba con el sistema completo conectado a la red cableada, recibiendo datos online. Los datos que recibía los iba imprimiendo por pantalla y estos iban siendo guardados en continuo en una base de datos **RRDTool** mediante un script. Mientras tanto cada cierto tiempo (tasa de refresco) los datos eran consultados de la base de datos RRDTool y sustituían los archivos que la página web mostraba. De esta forma al recargar la web los nuevos datos eran mostrados.

A continuación se muestra una figura de la interfaz finalizada. La captura original se ha recortado para que se pueda apreciar mejor.

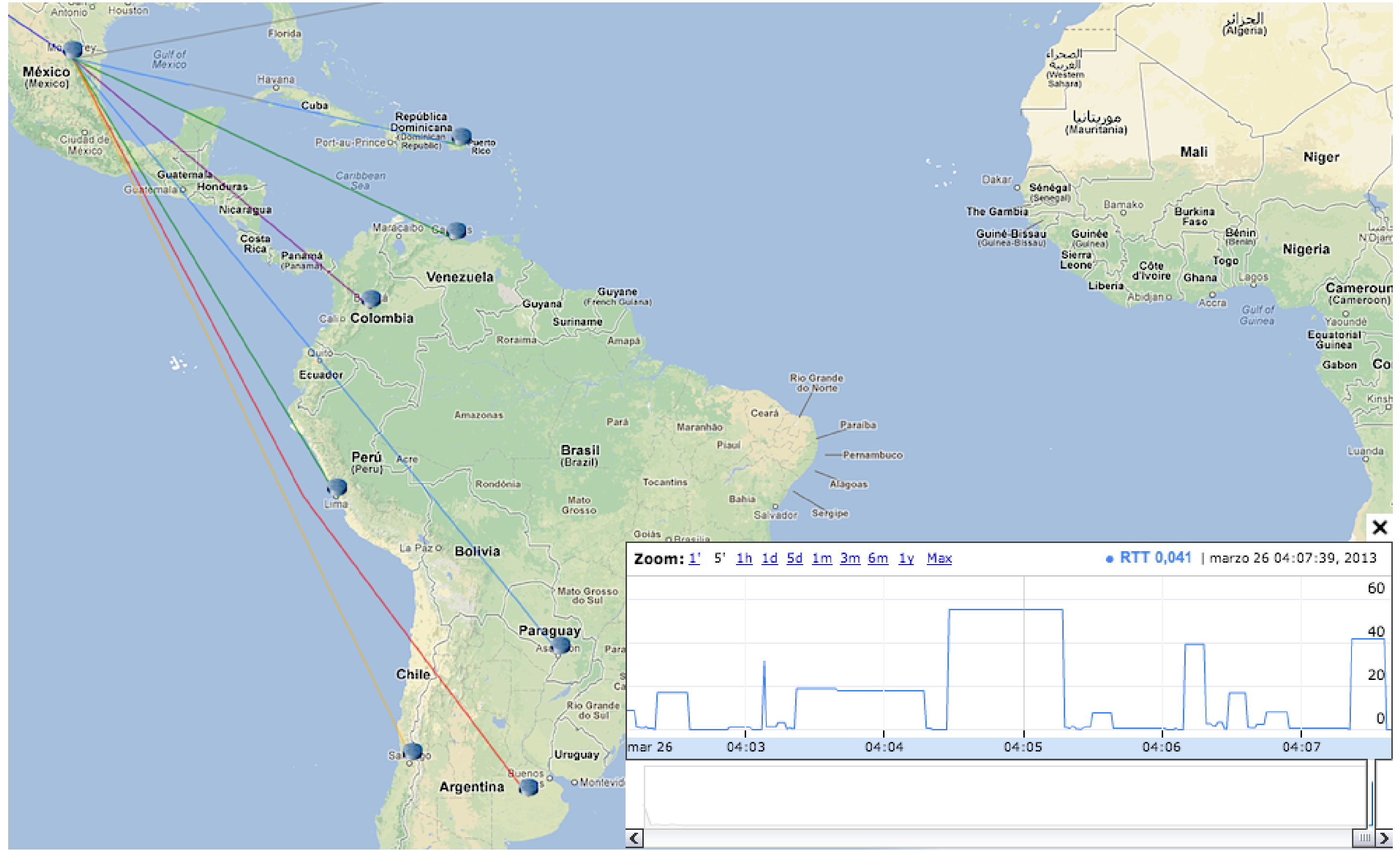

Figura 6.3: Interfaz web finalizada

6.5. Conclusiones

Las pruebas demuestran que el sistema es capaz de enfrentarse a situaciones propias de entornos de alto rendimiento y dar una respuesta rápida y segura. Tiene un uso eficiente de los recursos (Tiempo y memoria) y aprovecha las posibilidades de los sistemas donde se ejecuta (Multiproceso, velocidad de los discos duros, etc). Es estable y ofrece distintas formas de filtrado y una alta portabilidad.

La interfaz web permite mostrar varios gráficos a la vez que el mapa y dibujar una gran cantidad de puntos sin que afecte al rendimiento. Cumple con los requisitos funcionales y no funcionales especificados en el **Capítulo 2**.

7
Conclusiones

El principal punto fuerte de este proyecto es su aplicación en entornos reales con necesidades reales y la resolución de un problema real. El disector es muy estable y satisface los requisitos de los entornos de alto rendimiento. La aplicación es rápida, usa pocos recursos de memoria, aprovecha al máximo los recursos del sistema donde se ejecuta (Velocidad de los discos duros, procesamiento y paralelismo). Además se ha hecho un esfuerzo en portabilidad y reducción de dependencias para su uso en distintas plataformas.

La interfaz web provee una forma sencilla y usable que muestra los datos calculados con un rendimiento de dibujo más que suficiente para utilizar con comodidad y fluidez la interfaz.

A partir de los datos devueltos por el disector y las conclusiones de la visualización de estos en la interfaz web podemos tener una idea clara del estado de la red que estamos analizando. Además gracias a las distintas opciones de filtros podemos crear estadísticas y gráficos que nos permitan sacar conclusiones más específicas de la red.

Hay aspectos que pueden ser claramente mejorados en este proyecto tanto en el disector como en la interfaz web. Algunos como la velocidad de procesamiento donde se pueden optimizar aún más las estructuras de datos o aprovechar al máximo los núcleos de los procesadores modernos. La interfaz deja espacio para mostrar nuevos tipos de gráficos y estadísticas en el futuro.

En resumen, este proyecto resuelve un problema actual y real y propone distintas mejoras para el dearrollo futuro de nuevas características.

8

Mejoras futuras

8.1. Mejoras en el disector

El disector siempre podrá ser mejorado en cuanto a rendimiento, opciones y adición de nuevas características al surgir nuevos requisitos. Nuevos filtros, formatos de impresión con datos estadísticos o ser aprovechar las altas velocidades de las nuevas redes son unos buenos ejemplos.

8.1.1. Imprimir resumen por URL o dominio

Añadir una opción que al finalizar el análisis del fichero imprima un resumen con los retardos medios de cada URL o dominio y así poder realizar estadísticas o gráficos con estos datos. De esta forma podemos detectar un dominio que esté dando problemas en los tiempos de respuesta.

8.1.2. Realizar pruebas con redes de 10 y 40Gbps

Analizar cuántos paquetes pierde en redes de 10 y 40Gbps e intentar reducir el número para poder utilizar el programa en tiempo real en redes de este tipo.

8.1.3. Uso de las cabeceras HTTP

Se podría añadir una opción que tuviese en cuenta o imprimiese con los datos de la respuesta las cabeceras HTTP, todas o las que especificásemos por parámetro. De esta forma podríamos hacer un análisis más específico del tráfico HTTP. Es compatible con la propuesta del resumen por URL o dominio.

8.2. Mejoras en la interfaz web

La interfaz web también podrá ser mejorada siempre que haya nuevos requisitos o tecnologías que faciliten la visualización de los datos a mostrar o mejoren el rendimiento de los gráficos.

8.2.1. No utilizar Flash

Una de las mejoras más importantes es la de no utilizar una tecnología que requiera tantos recursos de CPU y que deba estar instalada por el usuario. El uso de tecnologías HTML5, canvas, svg y JavaScript que están incorporadas en los principales navegadores es una meta muy a tener en cuenta. De momento habrá que esperar a que se diseñen sistemas de gráficos potentes para estas tecnologías que permitan dibujar un gran número de puntos y gráficos sin afectar al rendimiento.

8.2.2. Uso de gráficos Retardo/Dominio

Añadir un gráfico asociado a los enlaces que muestre los retardos por dominio o URL para saber de un vistazo qué dominios están teniendo más problemas de retardos.

8.2.3. Gráficos en tiempo real

Permitir la actualización en tiempo real de los gráficos mostrados mediante transacciones AJAX o un servicio web que interactúe con la base de datos RRDTool. Hay funciones en apache que permiten consultar la base de datos. Con AJAX se pueden realizar peticiones asíncronas que actualicen los datos cada cierto tiempo.

8.2.4. Topografía de la red en los nodos de la interfaz web

Se podría asociar a cada nodo del mapa una representación de la topología red en la que los colores representan la carga de los enlaces (por ejemplo) o los enlaces con mayor nivel de retardos, por ejemplo.

Se podría también hacer un análisis topológico con las direcciones IP de origen y destino de las transacciones HTTP para mostrar mediante una imagen como la adjunta que muestre según el color los enlaces con mayor nivel de retardos.

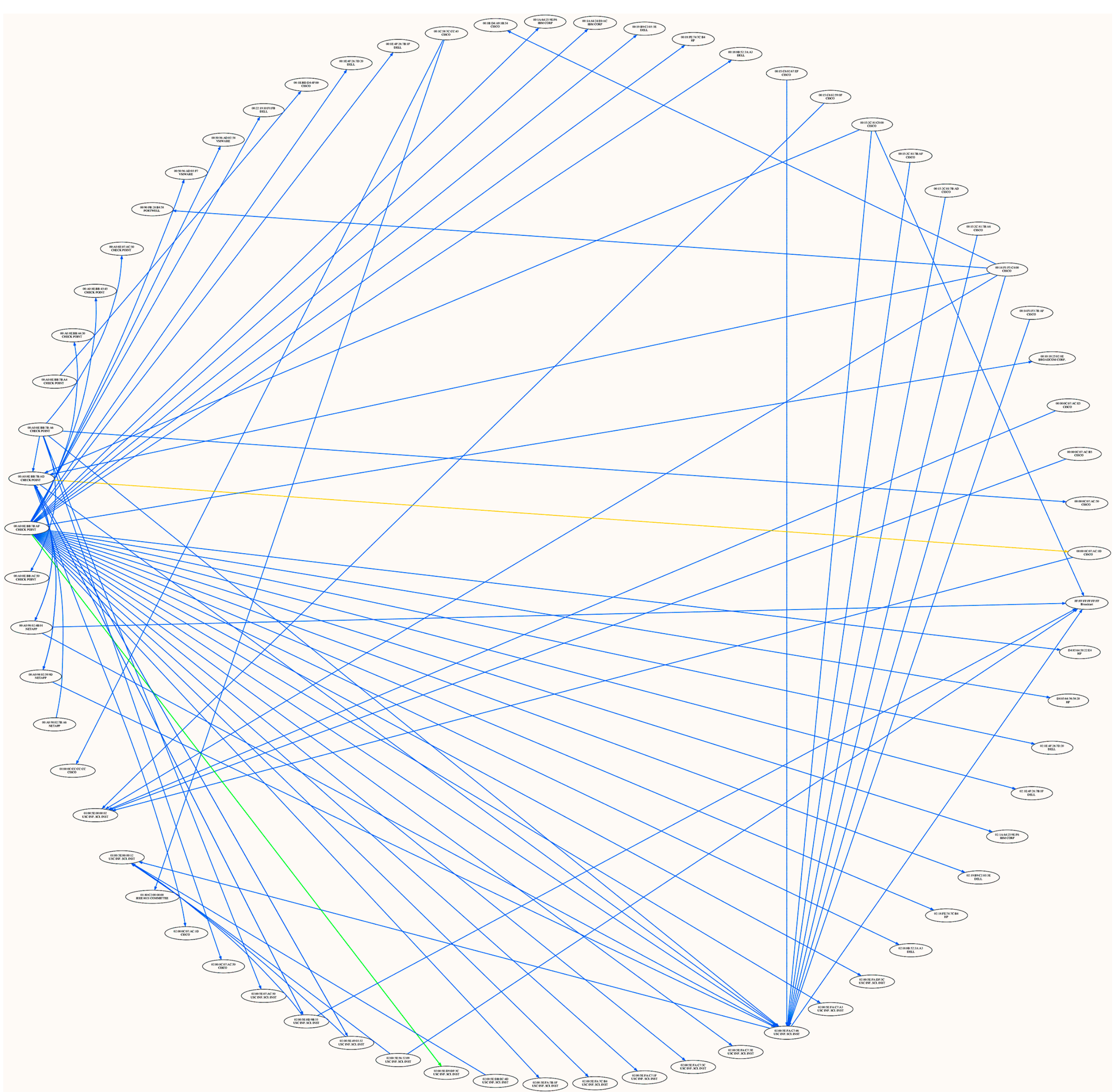

Figura 8.1: Ejemplo de topología de red

Bibliografía y Referencias

[1] **James F. Kurose y Keith W. Ross** Redes de computadoras : Un enfoque descendente basado en Internet.

[2] **Libnids** : http://libnids.sourceforge.net

[3] **Libpcap** : http://www.tcpdump.org/pcap.html

[4] **GLib** : https://developer.gnome.org/glib

[5] **RRDTool** : http://oss.oetiker.ch/rrdtool

[6] **Google Chart Tools** : https://developers.google.com/chart/?hl=es

[7] **Google Maps API** : https://developers.google.com/maps/?hl=es

[8] **Flot** : https://code.google.com/p/flot/

[9] **jQuery** : http://api.jquery.com