UNIVERSIDAD AUTÓNOMA DE MADRID

Escuela Politécnica Superior

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

AUTOMATIZACIÓN Y DETECCIÓN DE ANOMALÍAS EN TRÁFICO DE INTERNET

Eduardo Miravalls Sierra

Tutor: Javier Aracil Rico

Junio 2016

Automatización y detección de anomalías en tráfico de Internet

Autor: Eduardo Miravalls Sierra
Tutor: Javier Aracil Rico

HPCN Research Group
Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Junio 2016

RESUMEN

Resumen En las redes multigigabit de hoy en día, TCP/IPv4 es el protocolo de comunicación estándar. Para controlar la salud de la red, es habitual utilizar mecanismos de monitorización pasiva para controlar distintos tipos de indicadores de problemas que pueden sufrir las conexiones TCP. Algunos de estos son el desorden de los paquetes, la existencia de tráfico duplicado, clientes que anuncian ventanas de tamaño 0, exceso de conexiones que terminan con RST, o un elevado número de retransmisiones.

Precisamente, detectar retransmisiones en flujos TCP supone todo un reto, debido a que es difícil determinar si un paquete es una retransmisión o no midiendo en un punto intermedio de la red sin utilizar muchos recursos computacionales y de memoria. Se ha explorado el estado del arte y no se han encontrado soluciones capaces de funcionar en redes con tráfico a 10Gbps que cumplan estos requisitos.

El objetivo final de estos algoritmos es determinar si un flujo tiene muchas retransmisiones con respecto a la proporción de paquetes que pertenecen al mismo. Para lograrlo, en este trabajo se estudian dos algoritmos que determinar de forma heurística si un paquete es o no una retransmisión. Los algoritmos estudiados se centran en el seguimiento del número de secuencia más alto que se ha visto en un flujo, y en función de la posición relativa del siguiente segmento decidir si es una retransmisión.

Se ha desarrollado un programa que implemente ambos algoritmos para probarlos. Se han realizado pruebas con tráfico HTTP obtenido de entornos de producción. Las pruebas han sido tanto de validación contra un programa ya existente de análisis forense, como de rendimiento para comprobar que el algoritmo que daba mejores resultados se podría integrar en DetectPro, una aplicación comercial de análisis de redes a 10Gbps.

El resultado de las pruebas ha resultado ser exitoso para uno de los dos algoritmos, y ha resultado viable para ser probado en entornos de producción.

Palabras clave TCP, retransmisiones, análisis de red, monitorización, 10Gbps

Abstract

Abstract The TCP/IP protocol is the standard communication mechanism in nowadays multigigabit networks. Network administrators use passive monitoring solutions to check the network's health status. They usually look for an assortment of possible problems which may arise, such as duplicate packets, packet reordering, TCP hosts sending zero window announcements, an increase of RST packets, or retransmissions.

Incidentally, the focus of this work is to study why TCP retransmissions occur and how to detect them. Detecting TCP retransmissions is a challenging problem which has no simple accurate approach. Looking at the current state of the art, it seems there aren't any solution readily available which can accurately rule if a packet is a retransmission without performing complex trace analysis with high computational and memory requirements.

This work's ultimate goal is to check whether it is possible to easily test if a TCP flow had too many retransmissions. For this purpose, this work presents two heuristics to evaluate if a packet is or isn't a retransmission. Both algorithms keep track of the highest seen sequence number in a flow, and they decide if the following packet is a retransmission by looking to the partial ordering of the packet's sequence number and the aforementioned flow's current highest seen sequence number. Both algorithms should strive to give an informed hint of the approximate number of retransmissions a TCP flow had.

In order to test both algorithms, a small test program has been implemented and it has been fed real world HTTP traffic. The algorithm's output has been validated against an existing forensic analysis software. Unfortunately, only one of the algorithms gives correct results with a reasonable margin of error. That algorithm's performance has been tested, and has achieved 10Gbps when integrated in a commercial application.

I conclude that this work's objectives have been successfully achieved, and it's worth trying to deploy it in real word environments.

Keywords TCP, retransmissions, network analysis, monitoring, 10Gbps

Agradecimientos

A mis compañeros del laboratorio, que a pesar de los muchos problemas, me han estado echando un cable desde que entré en la laboratorio.

A mi tutor Javier Aracil y al grupo HPCN por dejarme trabajar para ellos y brindarme la oportunidad de desarrollar este Trabajo Fin de Grado.

A todos mis compañeros del doble grado por la ayuda que me han ido brindando a lo largo de estos 5 años.

Y a mis padres y mi hermana, que me han aguantado todos estos años.

«Rise and shine, Mister Freeman. Rise and... shine.
Not that I... wish to imply you have been sleeping on the job.
No one is more deserving of a rest, and all the effort in the world would have gone to waste until... well, let's just say your hour has... come again.
The right man in the wrong place can make all the difference in the world.
So, wake up, Mister Freeman. Wake up and... smell the ashes...»
G-Man

Índice general

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

Índice de Algoritmos

GLOSARIO

ACK asentimiento (ACKnowledgment). Dependiendo del contexto, se puede referir al valor del campo ACK en la cabecera TCP, o se puede referir a un paquete TCP sin datos cuyo único propósito es enviarle al otro equipo el asentimiento del último segmento que ha recibido. 3

Asentimiento ver ACK. 3, 6, 8

Falso Positivo en contraste de hipótesis, corresponde con el error de decir que la condición buscada se cumple cuando en realidad no se observa. XIX, 24

Falso Negativo en contraste de hipótesis, corresponde con el error de decir que la condición buscada no se cumple cuando en realidad si se observa. XIX, 24

FIN Es un tipo de segmento especial TCP con la bandera FIN de la cabecera TCP activada. Se envía al otro extremo para indicarle que no se enviarán más segmentos con datos. Debe ser asentido por el otro extremo. XVII, 6, 12, 14, 15, 20

Flujo TCP una conexión TCP consiste en un canal de comunicación bidireccional entre dos aplicaciones, que pueden o no estar en equipos distintos. Un flujo TCP es el conjunto de segmentos que conforman uno de los dos sentidos de dicha comunicación. 12, 25

Internet Protocol protocolo de la capa de red que permite la comunicación entre equipos. XIX

Longitud del buffer de captura número máximo de bytes que se almacenan en el fichero pcap por paquete. 15

Maximum Segment Lifetime Tiempo máximo que un segmento TCP debería existir en una red. XIX, 14

Round Trip Time Tiempo que tarda un paquete en viajar del emisor al receptor y de vuelta al emisor. XIX, 8

RST ReSeT. Es un tipo de segmento especial TCP con la bandera RST de la cabecera TCP activada. Indica al otro extremo que no hay ninguna conexión activa para ese puerto destino, y que no debe enviarle más segmentos. 1, 5, 6, 12, 14, 20

Segmento TCP conjunto de bytes que conforman la cabecera TCP y los datos que transporta. 2, 3, 4, 13

Sniffer programa que captura el tráfico de una red y permite su análisis para detectar problemas en la misma. 1

SPAN Switch Port ANalyzer, mecanismo de monitorización de tráfico. Consiste en que uno de los puertos de un switch reenvíe todo el tráfico que recibe por las otros puertos. 25

Transmission Control Protocol protocolo de la capa de transporte que proporciona un mecanismo de comunicación lógico entre procesos diferentes. XIX, 1

Traza fichero que almacena paquetes de red. 8

Ventana deslizante mecanismo que tiene TCP para controlar cuáles son los bytes que el otro extremo ha asentido, los bytes que se han enviado pero todavía no se han asentido, y los bytes que todavía no se han enviado. 3, 5

Acrónimos

EPS Escuela Politécnica Superior. 18

FN Falso Negativo. *Glosario:* Falso Negativo, XIX, 24, 25, 26

FP Falso Positivo. *Glosario:* Falso Positivo, XIX, 24, 25, 26, 31

GRSST Grupo de Redes, Sistemas y Servicios telemáticos. 8, 18

HPCAP High Performance CAPture. 9

HPCN High Performance Computing and Networking. 18, 27

HTTP Hypertext Transfer Protocol. 23

IP Internet Protocol. *Glosario:* Internet Protocol, XIX

ISN Initial Sequence Number. 3, 13

MSL Maximum Segment Lifetime. *Glosario:* Maximum Segment Lifetime, XIX, 14

NIC Network Interface Card. 27

NUMA Non Uniform Access Memory. 24, 27

PCAP Packet CAPture. 18

RAID Redundant Array of Independent Disks. 27, 28, 29

ReTx retransmisiones. 12, 21, 22

RTT Round Trip Time. *Glosario:* Round Trip Time, XIX, 8

SACK Selective ACK. 8

TCP Transmission Control Protocol. *Glosario:* Transmission Control Protocol, XIX, 1, 13, 14, 16

UAM Universidad Autónoma de Madrid. 18

UPNA Universidad Pública de Navarra. 8, 18

1 Introducción

1.1 Motivación

El protocolo Transmission Control Protocol (TCP) es omnipresente en las comunicaciones vía internet y es muy resistente a todo tipo de fallos y problemas, lo cual lo hace un protocolo complejo y presenta varios retos a la hora de monitorizar el estado de las conexiones mediante un sniffer.

Uno de los principales indicadores de problemas en una red que se buscan al realizar un análisis de su estado, es la presencia de conexiones problemáticas, que presenten muchos anuncios de ventanas de tamaño cero, que terminen muchas conexiones con RST o que los flujos tengan muchas retransmisiones, entre otros problemas.

Detectar retransmisiones en flujos TCP en redes de altas prestaciones, con varios cientos de miles o millones de flujos TCP concurrentes, es un problema que requiere un alto coste computacional y de memoria.

Es por esto que el principal objetivo de este trabajo es estudiar y probar algoritmos para detectar flujos con suficientes retransmisiones como para que indiquen un problema en la red. Dichos algoritmos deben ser económicos en términos computacionales y en consumo de memoria para poder operar a tasa de línea en redes de alto rendimiento.

1.2 Alcance

Aunque hay muchos indicadores de problemas en la red, este trabajo se centra en las retransmisiones. Para ello, se proponen y estudian dos posibles algoritmos para la detección de retransmisiones en flujos TCP.

Se ha limitado el estudio a la detección de retransmisiones en flujos TCP IPv4, ya que el tráfico IPv6 todavía no es predominante en las redes comerciales.

En ocasiones, puede ocurrir que haya paquetes duplicados en la red que si no se tienen en cuenta, pueden parecer retransmisiones. Se asumirá que no hay paquetes duplicados debido a que es un problema ya estudiado [24] y existen mecanismos para eliminarlos como paso previo antes de evaluar si el paquete es o no una retransmisión. De hecho, el driver HPCAP [16], con el que se han capturado las trazas que se han usado en las pruebas de este proyecto, soporta durante la propia captura un mecanismo de eliminación de duplicados.

1.3 Objetivos

Los objetivos de este trabajo son los siguientes:

- Diseñar e implementar un prototipo de pruebas que permita probar diferentes algoritmos para la detección de retransmisiones.
- Determinar la fiabilidad de los algoritmos propuestos en la sección 3.2, estudiando si detectan retransmisiones, o no las detectan, o si detectan como retransmisiones segmentos que no lo son.
- Comprobar que dichos algoritmos se pueden implementar de forma que sus requisitos de memoria sean reducidos y que sea posible aplicarlos a tasa de línea en enlaces de 10Gbps.
- Integrar el algoritmo que obtenga los mejores resultados en una herramienta comercial que opera a 10Gbps y verificar que no provoca ningún impacto en su rendimiento.

1.4 TCP: transferencia de datos fiable

A continuación, a modo de introducción teórica del problema a resolver, se expone el mecanismo de transferencia de datos fiable de TCP y cómo se producen las retransmisiones.

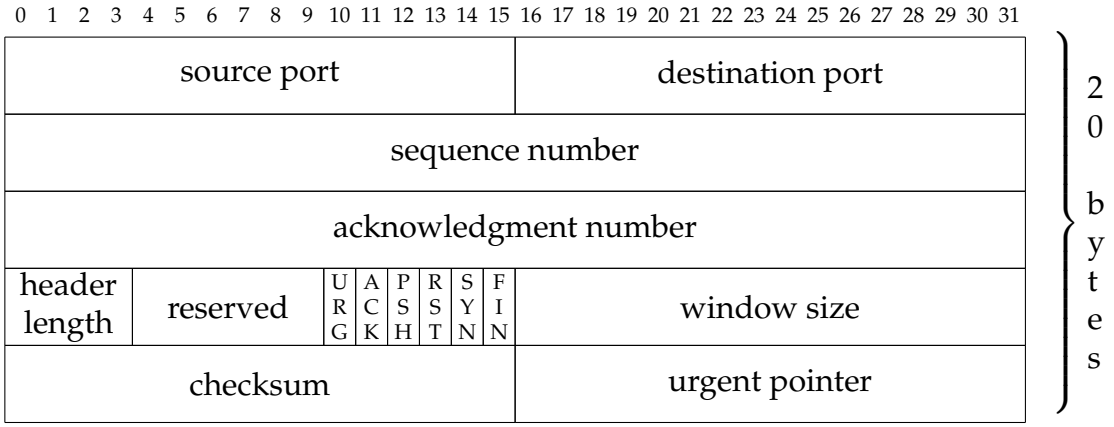

Figura 1.1: La cabecera TCP [21]

1.4.1 Números de secuencia TCP

Una de las características que tiene TCP es que garantiza que todos los bytes llegarán al destino de manera ordenada. Para ello, el protocolo TCP identifica cada byte de datos que envía un sentido de una conexión con un número de secuencia. Un número de secuencia es un entero de 32 bits que identifica de manera única a un byte hasta que el contador se "de la vuelta" (vuelva al valor inicial), lo que ocurre cada 2^{32} bytes de datos.

Por motivos de seguridad y para evitar que se puedan confundir segmentos de una conexión anterior, cada extremo de una conexión TCP suele empezar con un número de secuencia aleatorio o Initial Sequence Number (ISN), de manera que comparar números de secuencia no es una operación tan directa. La implementación final se explica en la sección 3.3.

Como se ilustra en la figura 1.1, TCP incluye un campo en la cabecera de cada segmento en el que indica el número de secuencia del primer byte de datos del segmento. TCP siempre envía cadenas de bytes consecutivas, de manera que con el número de secuencia y la longitud de los datos del segmento, se puede saber qué bytes se están enviando en un segmento.

1.4.2 Retransmisiones TCP

Para garantizar la entrega de los bytes, TCP utiliza asentimientos, o ACKs. Un ACK es un entero de 32 bits en la cabecera TCP que indican al otro extremo de la conexión que se han recibido todos los bytes cuyos números de secuencia sean menores que éste, o lo que es lo mismo, el siguiente número de secuencia que espera recibir.

De este modo, cada extremo de una conexión TCP debe mantener lo que se llama una "ventana deslizante" con la que llevar la cuenta de cuál es el último valor que ha asentido el otro extremo, qué bytes se han enviado que todavía no se han asentido, y qué bytes todavía no se han enviado.

Hay dos clases de retransmisiones ([7] p242-250), las ocasionadas por la expiración del temporizador y las generadas por el algoritmo "Fast Retransmit". El primer mecanismo consiste en mantener un temporizador de retransmisiones, con el que se controla si el otro extremo tarda demasiado en asentir segmentos. Cuando el temporizador expira, se reenvía el segmento con el número de secuencia más pequeño sin asentir, y se reinicia el temporizador.

A medida que TCP recibe segmentos, va respondiendo ACKs. Si TCP no tiene datos que enviar, envía un paquete solo con la cabecera. Cuando recibe un segmento fuera de orden, responde de manera inmediata con un ACK del número de secuencia que esperaba recibir. Por tanto, recibir varios ACKs con el mismo número de secuencia es un indicador de que el otro extremo está recibiendo el tráfico desordenado o que se ha producido una pérdida, como se ilustra en la figura 1.2.

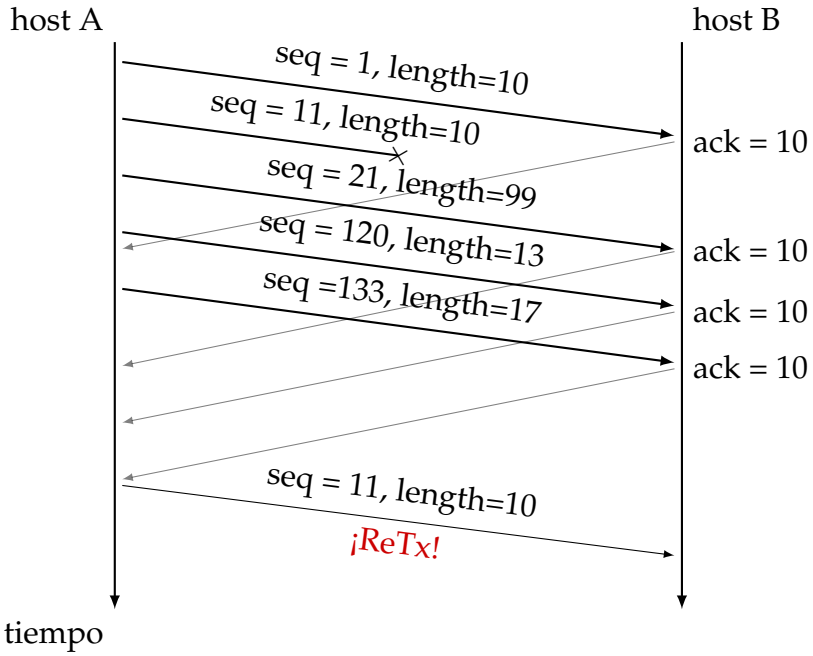

Figura 1.2: Ejemplo de una retransmisión

Cuando TCP recibe tres o más ACKs para el mismo número de secuencia, retransmite ese segmento que los ACKs del otro extremo le indican. Este algoritmo se conoce como "Fast Retransmit" [23]. TCP no tiene mecanismos para distinguir ACKs duplicados por desorden de los generados por una pérdida. Por este motivo, si el tráfico presenta mucho desorden, TCP realizará retransmisiones innecesarias, malgastando recursos de red.

Como se explica en [22], TCP es un protocolo muy flexible y permite que se produzcan las siguientes situaciones, que queremos ser capaces de detectar:

- Puede ser que un segmento se retransmita dividido en varios.
- Puede ser que varios segmentos se combinen y se retransmitan en uno solo.
- Puede enviarse un nuevo segmento, que incluya parte de dos segmentos, pero que no sea la totalidad de ninguno de los dos.

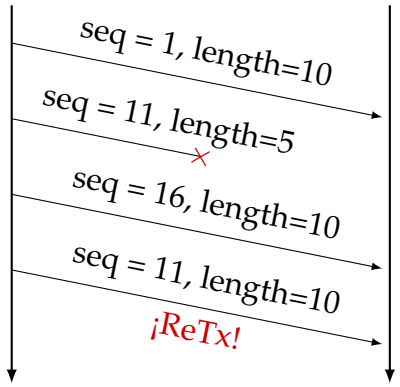

Figura 1.3: Ejemplo de solapamiento entre dos segmentos

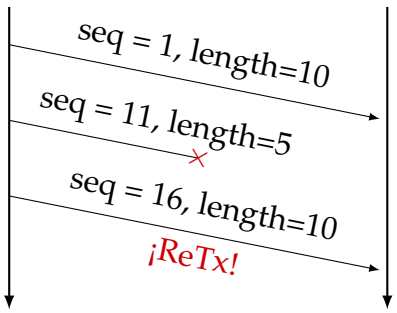

Figura 1.4: Ejemplo de retransmisión combinada con nuevos datos

1.4.3 Keep-Alives TCP

Un extremo de la conexión que lleve "suficiente" tiempo[1] sin recibir paquetes del otro extremo, puede enviar paquetes al otro extremo que le obliguen a contestar para comprobar que la conexión sigue activa.

La forma habitual de implementar esta funcionalidad (ver 4.2.3.6 de [5]) es enviar un byte fuera de la ventana deslizante del otro extremo, obligándole a enviar un ACK como respuesta. Como este paquete tiene como único objetivo que el otro extremo envíe una respuesta, es importante que el otro extremo no confunda un segmento con un byte válido con un Keep-Alive.

Si en lugar de recibir un ACK recibe un RST, TCP sabe que ha habido algún problema y el otro extremo ha terminado la conexión y por tanto, debe indicar a la aplicación que hubo un error y liberar también los recursos que reservó.

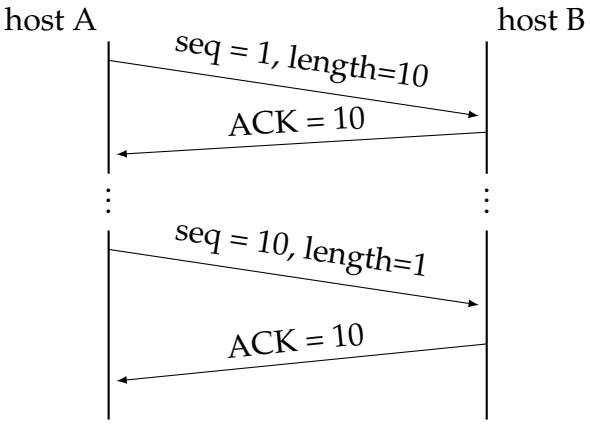

Figura 1.5: El host A envía un Keep-Alive al otro host, que le contesta con un ACK

[1] la definición de suficiente depende de la implementación

1.4.4 Finalización de una conexión TCP

El cierre normal clásico de la conexión TCP es un proceso de cuatro pasos (ilustrado en la figura 1.6) aunque, dependiendo de la implementación, se puede resumir a tres[2]. Este proceso tiene varios casos especiales [8], entre los que se incluyen el cierre parcial de uno de los extremos, que los dos extremos inicien el proceso a la vez, o puede incluso que uno de los extremos haya sufrido un error y responda con un RST.

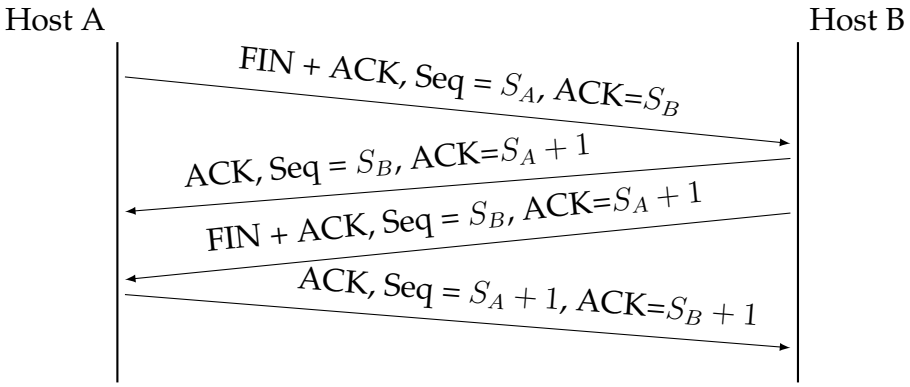

Figura 1.6: Cierre habitual de una conexión TCP

Debido a que el paquete FIN debe ser asentido, hay que tener en cuenta que consume un número de secuencia a pesar de no llevar datos. Esto es importante, ya que durante las pruebas de validación se han visto casos de conexiones en las que un extremo inicia el proceso de cierre enviando el FIN, y si el otro extremo no responde, lo retransmite varias veces. Tras un cierto periodo de espera, el equipo puede intentar retransmitir el FIN o puede que envíe Keep-Alives, cuyo número de secuencia puede depender del FIN anterior[3].

1.5 Organización del documento

En el capítulo 2 se comentan las líneas actuales de investigación acerca de las retransmisiones y los programas que se han usado para analizar tráfico de red en este trabajo.

En el capítulo 3 se expone el diseño de alto nivel del programa implementado, y se detallan los algoritmos utilizados, incluyendo los de detección de retransmisiones.

En el capítulo 4 se dan los detalles de implementación del programa de pruebas de acuerdo con el análisis realizado en el capítulo anterior.

En el capítulo 5 se pormenorizan las pruebas realizadas, el entorno en el que se han realizado, los resultados obtenidos, y se extraen algunas conclusiones de los mismos.

Finalmente, en el capítulo 6 se incluyen algunas reflexiones finales y líneas de trabajo futuro.

[2] el ACK que el host B envía en la figura se puede obviar
[3] También puede depender de la implementación.

ESTADO DEL ARTE 2

El problema de las retransmisiones TCP tiene dos enfoques: la detección y la prevención. El primer enfoque consiste en desarrollar mecanismos de monitorización pasiva de la red que permitan la detección de las retransmisiones. El segundo consiste en aplicar modificaciones y mejoras al protocolo o a los algoritmos que utiliza, para intentar maximizar el aprovechamiento del canal y reducir las retransmisiones innecesarias.

2.1 Detección de retransmisiones

Se han realizado varios esfuerzos por desarrollar programas de análisis de tráfico de red, aunque no parece ser viable su despliegue en entornos comerciales de alto rendimiento debido a que requieren de muchos recursos computacionales o de memoria. Los análisis suelen hacerse observando características de los paquetes de un mismo flujo, o incluso ayudándose de los ACKs que el otro extremo envía para concluir si se están realizando retransmisiones. Por otro lado, hay que considerar que las técnicas de análisis, cuanto más complejas, no solo aumentan los requisitos computacionales, sino que pueden necesitar que los flujos presenten características muy concretas para funcionar. Por ejemplo en [25] se asume que las pérdidas solo se producen a partir de un punto concreto de la topología de red que monitorizan, y que el tráfico no presenta desorden. Además, solo se detectan retransmisiones si en la traza están tanto el paquete original como la retransmisión. En [15] se propone una herramienta de análisis y generación de estadísticas, Tstat, que parece tener bajos requisitos de velocidad de procesador y memoria, pero solo se ha probado con datos extraídos de enlaces de baja velocidad y no parece una solución escalable.

Para intentar mejorar el rendimiento y ser capaces de procesar enlaces multigigabit, se puede intentar explotar las arquitecturas multicore modernas, o delegar parte de la carga computacional en hardware especializado. Es el caso de [22], donde se detalla un programa de generación de flujos TCP con la capacidad de detectar retransmisiones acelerado con tarjetas gráficas.

2.2 Prevención de retransmisiones espurias

Hay muchas publicaciones que proponen formas de reducir las retransmisiones espurias de TCP introduciendo extensiones de TCP y nuevos algoritmos. Las soluciones basadas en extensiones (añadir unos campos adicionales a la cabecera TCP) tienen todas el inconveniente de que añaden una pequeña sobrecarga adicional en todos los segmentos, y que solo se pueden utilizar si los dos extremos de la conexión la implementan.

Algunas de esas soluciones utilizan Selective ACK (SACK) [14], una extensión de TCP que permite asentir segmentos individuales. Al producirse un hueco en el buffer de recepción de un host B, esta extensión le permite al host A discernir si el hueco es de uno o varios segmentos, de manera que se realice el número mínimo de retransmisiones.

Otras se basan en añadir una marca de tiempo (timestamp) [9] en cada paquete. Esto permite estimar con mayor precisión el Round Trip Time (RTT) y distinguir un ACK de un segmento del ACK de su retransmisión, ya que la retransmisión tendrá una marca de tiempo posterior.

2.3 Herramientas utilizadas

A continuación se explican algunas de las herramientas utilizadas en este proyecto para analizar tráfico de red.

2.3.1 Wireshark/tshark

Wireshark [1] es la herramienta estándar de-facto para el análisis de trazas de pequeño tamaño.

Dispone de un amplio número de analizadores (más de 1350) de protocolos de red, y de una interfaz gráfica que permite inspeccionar los paquetes de manera interactiva.

tshark [2] es otra versión de esta misma herramienta, que permite el análisis de tráfico escribiendo en texto plano la información que se desee extraer de los paquetes.

Un gran inconveniente de estas herramientas es que requieren mucha memoria ya que cargan completamente en memoria la traza, lo que para trazas de más de un par de GB puede suponer requerir más memoria de la que dispone el equipo. Además, el elevado número de analizadores de los que disponen hace que requieran de muchísimo tiempo de proceso para analizar una traza. Y aunque se desactiven la mayoría de los analizadores, Wireshark puede requerir mucho tiempo de cómputo para analizar trazas de mayor tamaño.

2.3.2 ProcesaConexiones

Es una herramienta de análisis forense de grandes trazas de red. Está desarrollada por Daniel Morató Osés miembro del grupo de investigación Grupo de Redes, Sistemas y Servicios telemáticos (GRSST) de la Universidad Pública de Navarra (UPNA). Esta herramienta realiza un análisis exhaustivo de una traza a nivel TCP, sacando toda clase de estadísticas por conexión. Es de gran interés

para este trabajo ya que, entre otras cosas, se ha utilizado como base contra la que evaluar los algoritmos. Entre sus inconvenientes está en que tiene requisitos elevados de memoria, y que no puede trabajar a tasa de línea analizando un enlace de 10 Gbps.

Este programa lleva una lista de los intervalos de números de secuencia que se ha visto, y cada vez que se recibe un nuevo paquete, comprueba si su intersección es o no vacía con la lista de intervalos de números de secuencia vistos. Si la intersección es vacía, son datos nuevos, y si no, el segmento contiene datos retransmitidos. Si se está monitorizando un enlace que tiene pérdidas antes del punto donde se captura, ProcesaConexiones no será capaz de detectar las retransmisiones. Es por esto que el número de paquetes que reporta que contenían retransmisiones depende del punto donde esté colocado la máquina que capturaba el tráfico, ya que solo contabiliza aquellos que con mucha probabilidad son retransmisiones.

2.3.3 DetectPro

DetectPro [19] es una herramienta comercial multihilo, que realiza de manera simultánea las siguientes tareas:

- Captura el tráfico entrante. Para reducir los requisitos de espacio y de velocidad de escritura de los discos, solo escribe los primeros bytes de cada paquete, de manera que se pueda realizar un análisis basado en las cabeceras;
- Genera registros resumen de los flujos TCP;
- Realiza un seguimiento de tráfico agregado según redes configurables y genera alarmas de las mismas cuando se producen anomalías en, por ejemplo, la cantidad de flujos concurrentes.

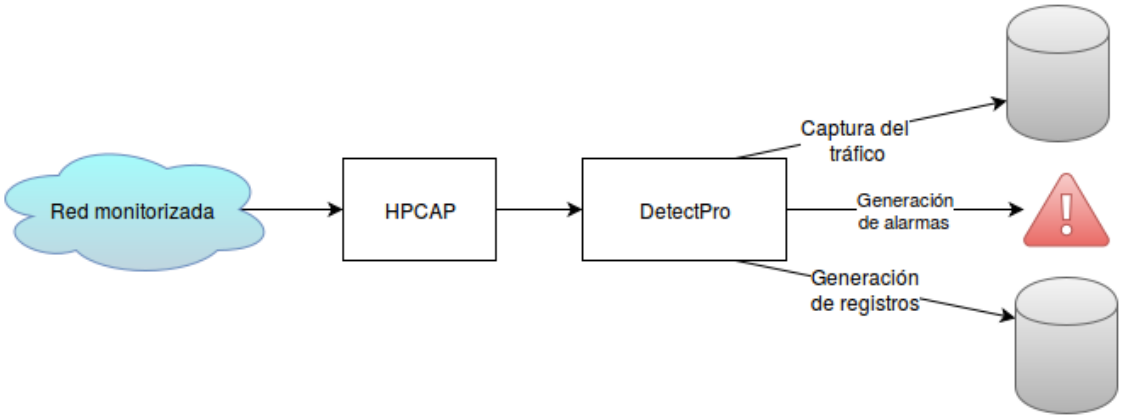

Figura 2.1: Esquema de las tareas que realiza DetectPro

DetectPro se ejecuta como aplicación cliente de High Performance CAPture (HPCAP), que es una versión modificada del driver del Intel capaz de capturar tráfico a 10Gbps, desarrollado por Víctor Moreno como su tesis doctoral [17].

2.4 Conclusiones

Observando el estado del arte, el proceso de análisis de flujos TCP para detectar retransmisiones en redes de alto supone un reto interesante de estudiar. Sin embargo, es un problema complejo y en este trabajo no se puede pretender encontrar la solución definitiva, ya que hay muchas variables que afectan al problema (desorden en los paquetes, que las pérdidas se produzcan antes del punto de captura, no observar todo el tráfico del flujo, entre otros), y hay que buscar un equilibrio entre lo que se desea detectar, las condiciones que se impongan al tráfico observado, y los requisitos de hardware.

3 Análisis y Diseño

En esta sección se presentan los algoritmos de detección de retransmisiones evaluados, cómo se analizó el problema, y cuál ha sido el diseño general del prototipo para probar dichos algoritmos.

3.1 Arquitectura del programa de pruebas

El objetivo final de este trabajo era evaluar los algoritmos de detección de retransmisiones, por lo que el diseño del programa debía ser sencillo para facilitar su depuración y mantenibilidad. Se optó por un diseño similar al de DetectPro, que permitiese seguir fácilmente la mecánica de procesamiento ilustrada en la figura 3.1.

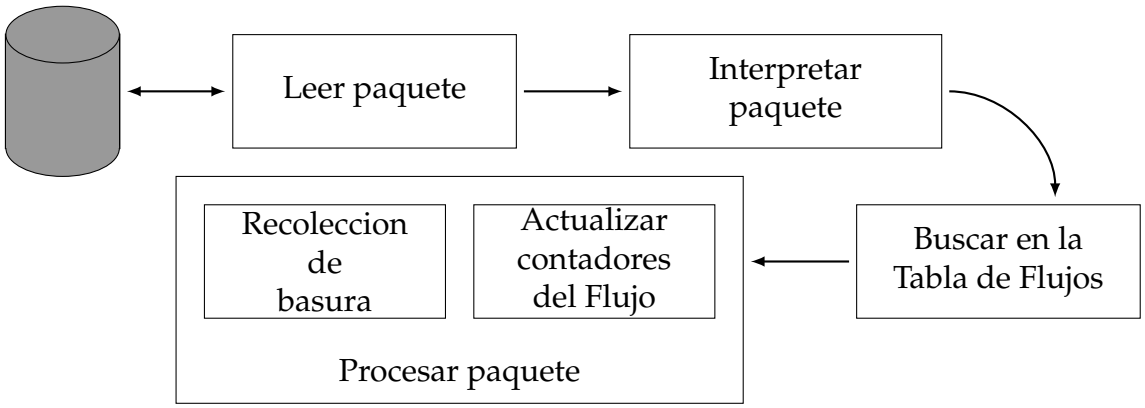

Figura 3.1: Esquema de la arquitectura del prototipo

1. En primer lugar debía tener un módulo que abstrajese la lectura de los paquetes y, a ser posible, que permitiese leer trazas en varios formatos. Esto se solventó mediante el uso de librerías, consultar la sección 4.2.1 para más información.

2. En segundo lugar se debía transformar la cadena de bytes a una representación interna en estructuras de datos. Esto facilita la lectura y mantenibilidad del código, y evita tener que usar desplazamientos en la cadena de bytes. Consúltese la sección 4.2.2 para los detalles de implementación.

3. En tercer lugar se debía encontrar un mecanismo eficiente para relacionar el paquete con el flujo al que pertenece, o crearlo en caso de ser un flujo nuevo. Éste módulo se bautizó "tabla de flujos" y su implementación se detalla en la sección 4.2

4. Y en ultimo lugar, habría que procesar el paquete. Procesar el paquete es una tarea compleja, ya que no es solo evaluar el algoritmo de detección de retransmisiones. Un paquete TCP podría tener activa la bandera de FIN o RST, que indican que la conexión TCP está terminando y que dentro de poco se podrán liberar los recursos de esa conexión (ver 3.4), o podría ser un paquete ACK o Keep-Alive que no deben ser confundidos con retransmisiones.

3.2 Algoritmos de detección de retransmisiones TCP

En este trabajo se han propuesto dos algoritmos para la detección de retransmisiones (ReTx) TCP, que se detallan a continuación:

3.2.1 Número de secuencia menor que el último visto

El primer algoritmo consiste en marcar como ReTx todo paquete cuyo número de secuencia esté por detrás del último número de secuencia visto.

Este algoritmo pasó por dos versiones. La primera versión, que se ilustra en pseudo-código en el Algoritmo 3.1, no detectaba ReTx parciales.

```
function IS_RETX_ALG1_V1(flow, pkt)
    return pkt.seq ≤ flow.last-seq
end function
```

Algoritmo 3.1: Pseudo-código inicial del algoritmo 1 para detectar retransmisiones

Como no es raro que haya ReTx parciales en un flujo, que el algoritmo no las detectase era un problema que se debía subsanar. La solución consistió en considerar que la cabecera TCP contiene el número de secuencia del primer byte que aparece en el paquete, y que, con la suma de ese valor más la longitud de la carga del paquete, se puede calcular el número de secuencia que debería tener el siguiente paquete (`next-seq`). Por tanto, si un paquete posterior tiene un número de secuencia menor que `next-seq` puede que sea una retransmisión.

```
function IS_RETX_ALG1_V2(flow, pkt)
    return pkt.seq ≤ flow.next-seq
end function
```

Algoritmo 3.2: Pseudo-código del algoritmo 1 con el error corregido

3.2.2 Huecos en el espacio de números de secuencia

El segundo algoritmo consiste en marcar como ReTx todo paquete cuyo número de secuencia cree un hueco en el espacio de números de secuencia. Este algoritmo se ilustra en pseudo-código en Algoritmo 3.3.

```
function IS_RETX_ALG2(flow, pkt)
    return flow.next-seq < pkt.seq
end function
```

Algoritmo 3.3: Pseudo-código del algoritmo 2 para detectar retransmisiones

El objetivo final de este trabajo no es determinar a ciencia cierta si un paquete es una retransmisión o no, sino detectar flujos que tengan suficientes retransmisiones como para indicar que existe problema. La idea de este algoritmo es marcar como ReTx el paquete que crea un hueco en el espacio de números de secuencia, que deberá ser rellenado por un paquete posterior, el cual será la retransmisión real.

Los resultados de las pruebas de validación apuntan a que este algoritmo no logra una estimación correcta del número de retransmisiones. Quizá esto sea debido a que no es capaz de distinguir un hueco de más de un paquete del hueco de un solo paquete, y esto lo haga muy conservador en sus estimaciones del número de paquetes retransmitidos. O quizá, debido a que es habitual que en las redes comerciales complejas haya cierto desorden en los paquetes de un flujo (por presencia de balanceadores de carga o si los paquetes toman caminos diferentes), puede ser que se produzcan huecos que rellenen paquetes posteriores, sin que se haya producido ninguna retransmisión.

3.3 Algoritmo para comparar números de secuencia

Comparar números de secuencia no es una tarea trivial ya que presenta varios retos:

- El espacio de números de secuencia es finito modular con 2^{32} valores posibles, y si se envían suficientes datos, se "da la vuelta", es decir, el siguiente número a $2^{32} - 1$ es 0.
- Como se explica en la sección "Initial Sequence Number Selection" de [21], el ISN suele ser aleatorio para evitar aceptar segmentos de una conexión anterior que ya terminó incluso aunque TCP pierda el estado y no sepa qué números de secuencia usó en sesiones anteriores. Por tanto, los números de secuencia pueden darse la vuelta tan pronto como llegue el segundo paquete del flujo.
- Al darse la vuelta, no se puede simplemente usar la comparación normal de enteros, ya que aunque s sea numéricamente menor que t, en el espacio de números de secuencia puede que sea al revés. Por ejemplo, si $s = 1000$ y $t = 2^{32} - 460$, tenemos que aunque $s < t$, en realidad s es el siguiente número de secuencia si el paquete t tenía 1460 bytes de carga.

function CMP_SEQ_LT(s, t)
  if $0 < (t - s) < 2^{31}$, computed in unsigned 32-bit arithmetic **then** $s < t$
  else $s \geq t$
  end if
end function

Algoritmo 3.4: Si s y t son números de secuencia, determina si $s < t$

En [21] no se da un algoritmo concreto para compararlos, pero en [9], en la sección "Protect Against Wrapped Sequence numbers (PAWS)" se presenta el pseudo-código para comparar números de secuencia, ilustrado en el Algoritmo 3.4. El algoritmo lo que hace es, dado un número de secuencia s, todo **número** t **menor** (siempre en aritmética módulo 2^{32}) que $s + 2^{31}$ lo considera **mayor que** s; y **si es mayor** que $s + 2^{31}$ lo considera **menor que** s.

Este algoritmo tiene el inconveniente de que solo funciona en caso de que la sesión TCP vaya a una tasa inferior a $16\ Gbps$, y para esos casos el intervalo de números considerados mayores que uno dado debería reducirse, aunque ésto lo haga menos resistente a desórdenes y a segmentos que por algún motivo perduren mucho tiempo en la red.

Para intentar ilustrarlo, recuperamos el ejemplo anterior de $s = 1000$ y $t = 2^{32} - 460$, donde tenemos que $s - t \mod 2^{32} = 1460$, es claramente menor que 2^{31}, y $t - s \mod 2^{32} = 4294965836$, que es mucho mayor que 2^{31}, y por tanto, $t < s$.

3.4 Gestión de la memoria: recolección de basura

Debido a que es posible que no se encuentre en la captura el paquete FIN o RST, se estableció un temporizador de expiración de flujo para volcar a disco el registro y liberar los recursos asociados de los flujos que lleven inactivos demasiado tiempo. Es un parámetro configurable del programa, que se ha fijado en 300 segundos para las pruebas.

Además, no se puede liberar un flujo en cuanto se encuentra un paquete con la bandera FIN o RST, debido a varios motivos:

- Pueden quedar segmentos en la red que todavía no hayan llegado al otro extremo que pertenezcan al mismo flujo.
- Debido a desorden en la captura o en la propia red, puede ser que observemos el paquete FIN o RST antes que otros paquetes anteriores.

Ésto ya se tuvo en cuenta en el diseño original de TCP [21], donde se define que el tiempo máximo que puede permanecer un segmento TCP en la red es el Maximum Segment Lifetime (MSL), este valor suele ser de unos 120 segundos, y se recomienda esperar dos MSL antes de suponer que todos los paquetes de una conexión han desaparecido de la red[1]. Sin embargo, se han observado casos en los que ese tiempo no es suficiente, y por ello, se ha dejado como un parámetro configurable. En las pruebas se ha fijado en 150 segundos.

[1] para el caso en el que un paquete tarde mucho en llegar a destino y su respuesta también.

3.5 Algoritmo para procesar cada paquete

En esta sección se presenta y explica el pseudo-código de la función encargada de procesar cada paquete en el Algoritmo 3.5. En dicho pseudo-código se utilizan las siguientes variables:

- bytes: la cadena de bytes que representa un paquete tal y como viaja por la red.
- header: estructura de metadatos del paquete. Contiene la marca de tiempo cuando llegó el paquete (`ts`), la longitud del buffer de captura (`caplen`) y la longitud de la cadena de bytes (`len`).
- `STALE_AFTER_SEC`: número máximo de segundos que hay que esperar antes de que expire un flujo por inactividad.
- `MSL`: número máximo de segundos que se estima que un segmento puede permanecer en la red antes de ser descartado por algún router o que llegue a destino.
- `ALGORITHM`: variable que indica qué algoritmo utilizar para marcar paquetes como retransmisiones.

```
 1: function PROCESS_PACKET(bytes, header)
 2:     EXPIRE_STALE_FLOWS( header.ts, STALE_AFTER_SEC )
 3:     REMOVE_FINISHED_FLOWS( header.ts, MSL )
 4:     pkt ← PARSE_PACKET(bytes, header)
 5:     if PKT_NOT_TCP( pkt ) then
 6:         return IGNORE_PACKET( )
 7:     end if
 8:     flow ← LOOKUP_HASHTABLE(pkt, HashTable)
 9:     if flow = NULL then
10:         INSERT_IN_HASHTABLE( pkt, HashTable )
11:     else
12:         UPDATE_LAST_ACCESS_TS(flow, header.ts)
13:         if IS_FIN(pkt) || IS_RST(pkt) then
14:             FINISH_FLOW( flow )
15:         else if IS_NOT_KEEPALIVE(pkt) && IS_NOT_ACK(pkt) &&
16: IS_RETX(pkt, ALGORITHM) then
17:             flow.numReTx++
18:         end if
19:         UPDATE_LAST_SEQ(flow, pkt)
20:     end if
21: end function
```

Algoritmo 3.5: Algoritmo de procesado de un paquete

A continuación se detalla el pseudo-código del algoritmo 3.5:

2-3: Utilizando como valor de "tiempo actual" la marca de tiempo del paquete actual, se hace una pasada de recolección de basuras. Esto se explica en la sección 3.4 y los detalles para una implementación eficiente se describen en la sección 4.2.4. No se puede usar como tiempo actual la hora de la máquina debido a que puede ser que se procesen paquetes a una velocidad mayor o menor de la velocidad a la que llegan según la traza. Esto supondría que el momento en el que se expira un flujo podría depender de la potencia de la máquina en la que se está ejecutando el programa, o el estado de carga en el que se encuentra, lo cual no es deseable.

4: En segundo lugar, hay que convertir la cadena de bytes a una representación interna en estructuras de datos para que el código sea fácil de leer, entender y mantener, sea lo más eficiente posible y no requiera tener que usar desplazamientos en la cadena de bytes (ya que es fácil introducir errores de programación difíciles de depurar). Consúltese la sección 4.2 para los detalles de la implementación.

5-6: En tercer lugar, se comprueba si el paquete es TCP/IPv4 y si no lo es, se descarta y se continua con el siguiente paquete.

8-10 A continuación se comprueba si el paquete pertenece a un nuevo flujo TCP o a un flujo ya existente:

- Si corresponde a un nuevo flujo, se crea el flujo y se almacena en la tabla hash.
- Si corresponde a un flujo existente, seguimos procesando

13-14 En quinto lugar, si es un paquete que marca el inicio de la fase de finalización de la sesión TCP, se debe marcar el flujo para su posterior recolección de basura.

15 Después se comprueba que el paquete no sea un ACK o un Keep-Alive ya que estos paquetes no se consideran retransmisiones.

16-19 En último lugar, se comprueba si el paquete es finalmente o no una retransmisión y se actualiza el contador de último número de secuencia visto.

DESARROLLO 4

4.1 Introducción

En esta sección se detalla cómo se ha desarrollado este trabajo, cómo se ha implementado la aplicación, y cómo se han resuelto los problemas que se explicaron en secciones anteriores.

4.1.1 Ciclo de vida

Para el desarrollo de la aplicación se escogió el ciclo de vida iterativo ya que permite ir haciendo sucesivas versiones, incluyendo cada vez mejoras al proyecto. Tras cada versión del programa, ésta se probaba tal como se detalla en el capítulo 5 y se analizaban los resultados obtenidos. Tras éste análisis, se revisaba el diseño realizado y se le aplicaban las mejoras posibles, que más tarde se implementaban y se volvían a probar.

4.2 Implementación

Se escogió C como lenguaje de programación debido a que es un lenguaje de bajo nivel que proporciona muy buen rendimiento, dispone de muchas librerías, y permite la gestión manual de la memoria.

La gestión manual de la memoria puede parecer un problema ya que es trabajo añadido, pero precisamente es una ventaja porque se pretende que la aplicación sea capaz de tratar con grandes volúmenes de datos y la gestión manual permite una mejor optimización. Otros lenguajes de alto nivel como Java o Python podrían requerir de mucha memoria en función de cómo se haga el diseño de las clases, y no sería fácilmente controlable.

4.2.1 Módulo de lectura

Para implementar este módulo se han utilizado las siguientes librerías:

libpcap

La libpcap es una librería para la captura, almacenamiento y lectura desde fichero de tráfico de red. Para el almacenado de los paquetes utiliza un formato que se ha estandarizado llamado Packet CAPture (PCAP). Más información disponible en [4] o en sus páginas del *man* de Linux.

NDLeeTrazas

Esta librería encapsula la lectura y escritura de ficheros en formato PCAP o RAW (ver [16] para una descripción del formato). Fue desarrollada por el equipo de investigación High Performance Computing and Networking (HPCN) de la Escuela Politécnica Superior (EPS) de la Universidad Autónoma de Madrid (UAM) y por el grupo GRSST de la UPNA.

NDLeeTrazas implementa una interfaz similar a la de la libpcap: tras abrir una o varias trazas, basta con invocar `NDLTloop` (una función similar a `pcap_loop`) con un puntero a la función que procesará cada paquete hasta que se hayan procesado todos.

4.2.2 Interpretación de las tramas

Para analizar las tramas de manera eficiente, se han utilizado las mismas estructuras que usa el Kernel de Linux en su pila TCP/IP.

Las estructuras, definidas en `if_ether.h`, `ip.h`, `tcp.h` y que se suelen encontrar en `/usr/include/netinet`, están diseñadas para que baste con hacer un casting del puntero al buffer a la estructura correcta para tener ya todos los campos identificados. Esto hace el código más legible, es eficiente y elimina errores de programación al calcular desplazamientos para encontrar dónde empieza un campo. Además, estas estructuras tienen en cuenta si la máquina es BigEndian o LittleEndian, lo que facilita la portabilidad del código.

```
1 // uint8_t *bytes es un puntero al principio del paquete
2 struct ethhdr *eth = (struct ethhdr *)bytes;
3 bytes += ETH_HLEN; // longitud en bytes de la cabecera Ethernet
4 struct iphdr *ip = (struct iphdr *)bytes;
5 struct tcp_header *tcp =
6         (struct tcp_header *)((uint32_t *)ip + ip->ihl);
```

Código 4.1: Se obvia el control de errores para abreviar

Hay que tener en consideración algunos detalles más, como que los paquetes pueden llevar uno o más tags VLAN ([7] p482-485), lo cual afecta tanto a la implementación del programa (puede haber o no una cantidad variable de cabeceras VLAN entre la cabecera ethernet y la IP) como a la hora de escribir filtros BPF para extraer conexiones concretas de trazas grandes para poder ser inspeccionadas con Wireshark.

Tampoco hay que olvidar la fragmentación IP [20], ya que solo contiene la cabecera TCP el primer fragmento. De este modo, el resto de fragmentos IP se pueden descartar ya que en este trabajo todo el análisis se centra en la cabecera TCP.

4.2.3 Tabla de flujos

Por cada paquete hay que buscar la información correspondiente al flujo al que pertenece. Si no se encuentra el flujo, hay que reservar una nueva estructura para dicho flujo, y añadirla a la estructura que almacena los flujos. Para poder buscar de forma eficiente tales estructuras se decidió utilizar una tabla hash.

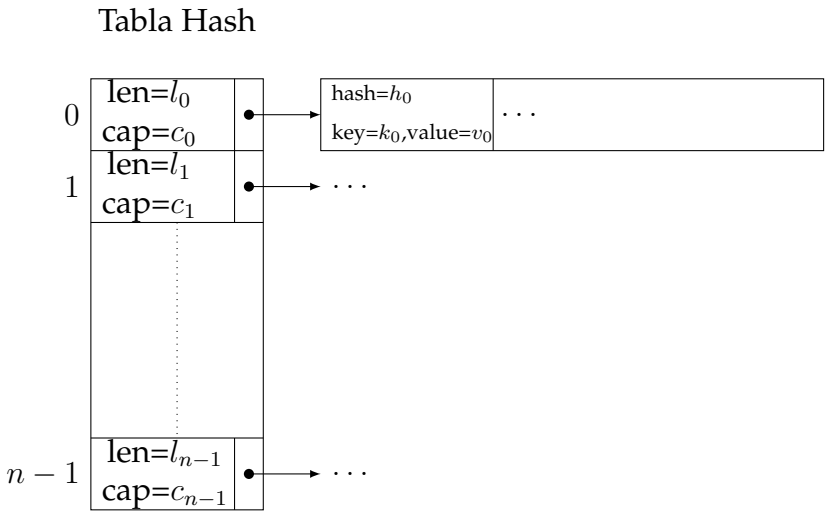

Figura 4.1: Esquema de la tabla hash implementada

La tabla hash implementada resuelve las colisiones mediante encadenamiento (*chained hashing*). Este método de resolución de colisiones consiste en que cada posición de la tabla (*bucket*) sea una lista en la que ir insertando los elementos que vayan a la misma posición. Dicha lista se ha implementado como un array de estructuras de tipo *entry*, ya que si la tabla es lo suficientemente grande, se espera que sean cadenas cortas, y si se implementan como una lista enlazada no se aprovecha el principio de localidad espacial que explota la caché de la CPU.

Identificador de un flujo

Para identificar un flujo TCP de manera única, se escogió como clave de la tabla de flujos la 5-tupla IP origen, IP destino, puerto origen, puerto destino, y el campo de protocolo de nivel superior de la cabecera IP. Estos valores suponen que un flujo TCP ocupe al menos 13 bytes en memoria (una IP son cuatro bytes, un puerto dos bytes, y el campo de protocolo es un byte).

Función hash utilizada

La función hash escogida es *One-at-A-Time* (código 4.2), diseñada por Bob Jenkins [10]. Se ha escogido porque tiene una serie de buenas propiedades, como son la de *avalanche* (entradas que difieren en muy pocos bits, generan hashes muy distintos), es relativamente eficiente ($O(9n + 9)$), y tiene pocas líneas de código. Otras alternativas requerían de muchas más líneas de código y no generaban resultados mucho mejores.

```
size_t BJ_OAT_hash(const char *key, size_t len) {
        size_t hash, i;
        for (hash = i = 0; i < len; ++i) {
                hash += key[i];
                hash += (hash << 10);
                hash ^= (hash >> 6);
        }
        hash += (hash << 3);
        hash ^= (hash >> 11);
        hash += (hash << 15);
        return hash;
}
```

Código 4.2: Código de la función hash utilizada

Mecanismo de búsqueda en la tabla hash

El programa se ha hecho de tal forma que aunque por cada paquete haya que computar el valor hash de su clave, solo se tenga que calcular una vez por paquete.

Una vez calculado el hash, se utiliza como índice en la tabla módulo el tamaño de la tabla y después se busca su clave en la lista de resolución de colisiones.

Para acelerar un poco las búsquedas, se ha comprobado que es más rápido comparar primero el valor hash del paquete y solo si coincide comparar las claves. Esto supone que cada entrada debe almacenar además el hash de la clave, porque si hubiese que recalcularlos cada vez que se busca, el rendimiento sería peor que simplemente comparar las claves.

4.2.4 Gestión de flujos activos/inactivos

Para realizar una gestión eficiente de los flujos activos, hay que implementar un mecanismo que no implique recorrer todos los flujos de la tabla para comprobar si alguno lleva demasiado tiempo inactivo. Para ello, lo que se ha hecho es mantener una lista enlazada de los flujos activos, ordenada según el último acceso. De este modo, el primer flujo de la lista será el último accedido, y el último flujo será el que lleve más tiempo sin ser accedido. Por tanto, se requería una lista doblemente enlazada que permitiese mover elementos rápidamente al principio de la misma, y que permitiese ser recorrida del último al primero.

Además, se debía mantener una segunda lista en paralelo que lleve los flujos en los que se ha visto un RST o un FIN, que, tras un tiempo de gracia, pueden ser liberados, asegurando además que un flujo no pudiese pertenecer a las dos listas a la vez.

Se optó por utilizar la lista enlazada del Kernel de Linux ([6] y [3]) ya que tiene la ventaja de que está altamente testada y que es una lista intrusiva. Una lista intrusiva es aquella que obliga a los elementos que se quieren insertar en ella a contener los enlaces de la propia lista, normalmente con un campo especial.

Esto satisface dos objetivos: en primer lugar, la lista no tiene que gestionar memoria (los dos punteros necesarios ya están en las propias estructuras que se quieren insertar), y en segundo lugar, esto obliga a que un flujo solo pueda pertenecer a una lista por cada campo.

La forma que tiene de implementarla el Kernel es con la estructura definida en el código 4.3.

```
1 struct list_head {
2         struct list_head *next, *prev;
3 };
```

Código 4.3: Definición de la lista del Kernel

`list.h` provee de todo un conjunto de primitivas para insertar y extraer nodos, y varias formas de recorrer listas. Además, tiene la peculiaridad de estar implementada como un conjunto de macros y funciones `static inline`. Esto supone un aumento del tamaño del ejecutable generado, pero en teoría eso se compensa generando un código más eficiente.

4.2.5 Descripción de la salida: estructura de un registro de conexión TCP

La salida del programa es un conjunto de valores separados por espacios, similar a la que produce ProcesaConexiones.

La ventaja de escoger este formato es que permite que la salida sea fácilmente procesada por programas de consola como AWK, o pueda ser importada a un programa que lea CSV utilizando como separador el espacio en blanco.

Por otro lado, un inconveniente que tiene este formato es que la salida no es estructurada y puede resultar confusa sin tener la documentación del programa. Tomando como ejemplo la siguiente línea:

```
10.248.130.157 50436 10.254.72.6 80 1393844004.968410000
1393844004.981758000 2 0 0 0 1 0 0 0 1
```

Los campos son

- la IP del origen (envió el primer paquete)
- el puerto origen
- la IP del destino
- el puerto destino
- marca de tiempo del primer paquete de la conexión
- marca de tiempo del último paquete de la conexión
- número de paquetes del origen hacia el destino

- número de paquetes del destino hacia el origen
- número de ReTx del origen hacia el destino
- número de ReTx del destino hacia el origen

Aunque no son estrictamente necesarios, los siguientes campos se incluyeron porque facilitan la labor de depuración sin necesidad de tener que lanzar el programa con gdb:

- número de paquetes sin datos del origen hacia el destino
- número de paquetes sin datos del destino hacia el origen
- número de paquetes keep-alive del origen hacia el destino
- número de paquetes keep-alive del destino hacia el origen
- un número indicando el motivo por el cual se generó el registro: 0 si se vio un nuevo SYN, 1 si el flujo lleva el suficiente tiempo inactivo, 2 si no hay más paquetes en la traza.

5 Pruebas y resultados

Las pruebas son una parte integral de este proyecto, ya que con ellas se evalúa tanto la corrección de la implementación (que el código haga lo que se espera que haga), la fiabilidad de los algoritmos (si los resultados que devuelven son significativos y transmiten cierta seguridad de que los datos sean correctos) y la eficiencia de los algoritmos (en términos de uso de memoria y cuánto tardan en procesar una traza).

5.1 Descripción de los datos de prueba

Para la realización de las pruebas se han tomado tres trazas con tráfico real anonimizado capturadas en diferentes entornos en los que predomina el tráfico web Hypertext Transfer Protocol (HTTP) cuyos datos se resumen en la tabla 5.1. De dichas trazas, se han descartado las conexiones TCP con menos de 100 paquetes entre los dos sentidos debido a que al tener pocos paquetes, cualquier métrica basada en proporciones produce resultados numéricamente correctos, pero de poca relevancia a la hora de detectar problemas. Por ejemplo, un flujo con 10 paquetes que tiene un 50 % de retransmisiones no es un flujo que resulte de gran interés a la hora de detectar problemas en la red.

Traza	Tamaño (GB)	Paquetes	Conexiones* TCP	Flujos* con ReTx
A	91GB	149 millones	98279	2972
B	120GB	211 millones	196580	705
C	387GB	539 millones	725648	13721

Tabla 5.1: Trazas utilizadas en las pruebas. (*) Conexiones con al menos 100 paquetes

5.2 Metodología de las pruebas

Para detectar flujos problemáticos, **se definió que** fue relevante la cantidad de retransmisiones de un flujo TCP si al menos tuvo 5 paquetes retransmitidos y de los paquetes enviados, un 5% fueron retransmisiones. Tal cantidad de retransmisiones comienza a ser relevante, y puede ser indicador de un problema en la red.

A la hora de evaluar la respuesta a esa pregunta, existen dos posibles errores que se pueden cometer. Es posible que el algoritmo que se está evaluando responda afirmativamente cuando la respuesta correcta sea que no, lo que se llama error de Falso Positivo (FP). O puede ser que responda negativamente cuando la respuesta correcta sea un sí, es decir, error de Falso Negativo (FN).

Para evaluar la salida de los dos algoritmos se tomó como resultado de referencia la salida de ProcesaConexiones y se determinó para cada registro si detectaba correctamente o no si tuvieron retransmisiones. Para poder identificar el registro de un flujo de manera única, se utilizaron la cuatro tupla y la marca de tiempo de inicio del flujo.

Esta forma de identificar los flujos tiene la ventaja de que si la cuatro tupla aparece varias veces entre los registros de salida de los programas, se pueden distinguir.

5.3 Pruebas de validación

El objetivo de estas pruebas es obtener resultados experimentales de la salida de ambos algoritmos, para ser contrastados y verificados contra ProcesaConexiones.

No se ha podido contrastar contra tshark debido a que el tiempo de cómputo y la memoria requerida eran descomunales. La traza C era demasiado grande como para que cupiese en memoria, y la traza B requirió de más de 300 horas de CPU. Además, debido a que en ocasiones puede no ser claro dónde termina un flujo y dónde empieza otro que reutiliza la cuatro-tupla (puede que en la traza no se vea el SYN pero sí un salto repentino en los números de secuencia de los dos sentidos) no se ha logrado cuadrar la salida del prototipo con la de tshark.

Por el contrario, ProcesaConexiones requirió de tiempos inferiores a una hora por traza, y en el caso del programa implementado el tiempo era de unos 10 minutos en el caso de la traza C, lo que permitía realizar las pruebas en poco tiempo.

5.3.1 Entorno experimental

Estas pruebas se ejecutaron en una máquina con una placa base Non Uniform Access Memory (NUMA) de Supermicro X10DRi-T4+ v1.01, con dos procesadores Intel®Xeon®CPU E5-2640 v3 a 2.60GHz, con 6 cores cada uno, y con 128 GB de RAM DDR3 a 2133MHz.

5.3.2 Resultados obtenidos con la traza A

En las tablas 5.2 y 5.3 se pueden ver los resultados obtenidos por ambos algoritmos. Estos resultados muestran que el segundo algoritmo obtiene resultados muy variados, y que no funciona correctamente.

Sin embargo, el primer algoritmo clasifica correctamente los flujos con un margen de error razonable.

	FN	Porcentaje FN	FP	Porcentaje de FP
Cliente a Servidor	0/1031	0 %	669/97248	0.69 %
Servidor a Cliente	0/1941	0 %	1360/96338	1.41 %

Tabla 5.2: Resultados de la traza A para el primer algoritmo frente a ProcesaConexiones

	FN	Porcentaje FN	FP	Porcentaje de FP
Cliente a Servidor	645/1031	62.56 %	16709/97248	17.18 %
Servidor a Cliente	1873/1941	96.50 %	932/96338	0.91 %

Tabla 5.3: Resultados de la traza A para el segundo algoritmo frente a ProcesaConexiones

Inspeccionando con Wireshark y analizando los registros correspondientes de los Falsos Positivos del algoritmo 1, parece que este caso de prueba tiene varios problemas:

- En algunas conexiones no se veía nada más que los paquetes de uno de los dos flujos. Esto suele deberse a un problema de configuración en el SPAN, que puede ser causado porque un flujo llevase tag VLAN y el otro no.
- En otros casos parece que haya pérdidas antes del punto de la red dónde se capturaba el tráfico. Esto se deduce de la presencia de flujos en los que el equipo A envía un paquete con un número de secuencia mayor que el que se esperaba el equipo B, seguido de varios ACKs duplicados del equipo B, para a continuación responder A con segmentos que llevan datos cuyos números de secuencia sean menores que el número de secuencia del primer paquete.
- En algunos flujos no parece que haya retransmisiones, pero si se observan ráfagas de segmentos desordenados. Por ejemplo: si se envían los paquetes n, $n+1$, $n+2$, pero en la captura están en el orden contrario $n+2$, $n+1$, n, el algoritmo 1 contará dos retransmisiones, que en realidad no lo son.
- También se observan varios casos de Falsos Positivos para conexiones que se encuentran al límite de la definición dada de conexión que tuvo retransmisiones. Por ejemplo, una conexión con 50 paquetes en un sentido y 60 en el otro, que ProcesaConexiones determine que tiene 4 retransmisiones en cualquiera de los dos sentidos, no tiene suficientes retransmisiones par cumplir la definición. Si, por otro lado, algoritmo 1 cuenta 5 paquetes como retransmitidos en cualquiera de sus sentidos, ese flujo tendría más de un $5\,\%$ de retransmisiones, por tanto se consideraría que tiene un número elevado de retransmisiones y se generaría un error de Falso Positivo.

5.3.3 Resultados obtenidos con la traza B

En las tablas 5.4 y 5.5 se muestran los resultados de los dos algoritmos frente a ProcesaConexiones en la clasificación de los flujos. En ellas, se ve que el segundo algoritmo marca la mayoría de los flujos como "no tienen retransmisiones", lo que provoca esa enorme cantidad de falsos negativos. Sin embargo, podemos ver que el primer algoritmo detecta correctamente todos los flujos que no tienen retransmisiones y, con un porcentaje de error muy bajo, detecta cuándo un flujo tiene retransmisiones.

	FN	Porcentaje FN	FP	Porcentaje de FP
Cliente a Servidor	0/171	0 %	38/196419	0.019 %
Servidor a Cliente	0/534	0 %	175/196056	0.089 %

Tabla 5.4: Resultados de la traza B para el primer algoritmo frente a ProcesaConexiones

	FN	Porcentaje FN	FP	Porcentaje de FP
Cliente a Servidor	152/171	88.89 %	30/196419	0.015 %
Servidor a Cliente	517/534	96.82 %	30/196056	0.015 %

Tabla 5.5: Resultados de la traza B para el segundo algoritmo frente a ProcesaConexiones

5.3.4 Resultados obtenidos con la traza C

En las tablas 5.6 y 5.7 se muestran los resultados de los dos algoritmos frente a ProcesaConexiones en la clasificación de los flujos. De nuevo, se observa que el primer algoritmo detecta casi sin error si un flujo tiene retransmisiones o no, y el segundo muestra porcentajes de error muy variados, destacando el caso de Servidor a cliente en la tabla 5.7, donde ha determinado que casi todos los flujos tuvieron retransmisiones, y por ello obtiene una tasa de FN muy elevada.

	FN	Porcentaje FN	FP	Porcentaje de FP
Cliente a Servidor	0/35	0 %	59/725613	0.008 %
Servidor a Cliente	0/13686	0 %	115/711962	0.016 %

Tabla 5.6: Resultados de la traza C para el primer algoritmo frente a ProcesaConexiones

	FN	Porcentaje FN	FP	Porcentaje de FP
Cliente a Servidor	5/35	14.29 %	95723/725613	13.19 %
Servidor a Cliente	13669/13686	99.88 %	91/711962	0.013 %

Tabla 5.7: Resultados de la traza C para el segundo algoritmo frente a ProcesaConexiones

5.3.5 Conclusiones de esta fase de pruebas

De los datos anteriormente expuestos se puede concluir que el segundo algoritmo no sirve para detectar de manera fiable retransmisiones, ya que obtiene tasas de ambos tipos de error elevadas.

Por otro lado, el algoritmo 1 presenta tasas de error muy reducidas, destacando que no comete falsos negativos, es decir, que cuando ha determinado que un flujo no tenía retransmisiones siempre ha acertado. Debido a su baja tasa de falsos positivos, se puede suponer que, con alta probabilidad, cuando ha determinado que un flujo tenía retransmisiones era cierto.

Esto lo hace un candidato viable para ser desplegado en entornos de análisis de red reales.

5.4 Pruebas de rendimiento

Una vez comprobada la fiabilidad del algoritmo 1, el siguiente paso lógico era evaluar la velocidad a la que procesa tráfico, para ver si es viable o no integrarlo en procesos a tiempo real en redes de alto rendimiento. Para ello, se integró el algoritmo 1 en DetectPro y se evaluó si el rendimiento del programa se veía sensiblemente reducido o no.

La prueba consistió en enviar la traza C (por ser la más grande) a 10 gigabits por segundo desde un equipo emisor a otro receptor que estaba ejecutando DetectPro.

La prueba duraba unos 335 segundos, y se repitió 20 veces tanto para el caso de DetectPro como para la versión modificada.

5.4.1 Entorno experimental

Para realizar esta prueba se usaron un par de servidores del grupo de investigación HPCN conectados mediante un cable de fibra óptica, uno con el rol de emisor de tráfico a 10Gbps y otro como receptor.

Para poder optimizar el rendimiento del sistema, es necesario conocer la arquitectura subyacente, y sobre todo en el caso de ser NUMA, ya que tiene algunas sutilezas: no solo hay que tener en cuenta que el hardware sea lo suficientemente potente, sino hay que saber cómo y dónde está conectado para configurarlo correctamente.

El servidor emisor contiene una NetFPGA que lee la traza C de un Redundant Array of Independent Disks (RAID) 0 de 8 discos SSD, y la envía a 10 Gbps por la fibra al servidor receptor. Esta generadora de tráfico es el resultado del TFG de José Fernando Zazo [26]. El servidor emisor tiene una placa Supermicro X10SL7-F, un procesador Intel®Xeon®E3-1230 v3 a 3.3GHz, y 32GB de RAM DDR3 a 1600MHz.

El servidor receptor tiene dos procesadores NUMA Intel®Xeon®CPU E5-2630 a 2.30GHz con 6 cores cada uno, montados en una placa Supermicro X9DR3-F y con 128GB de RAM DDR3 a 1333Mhz.

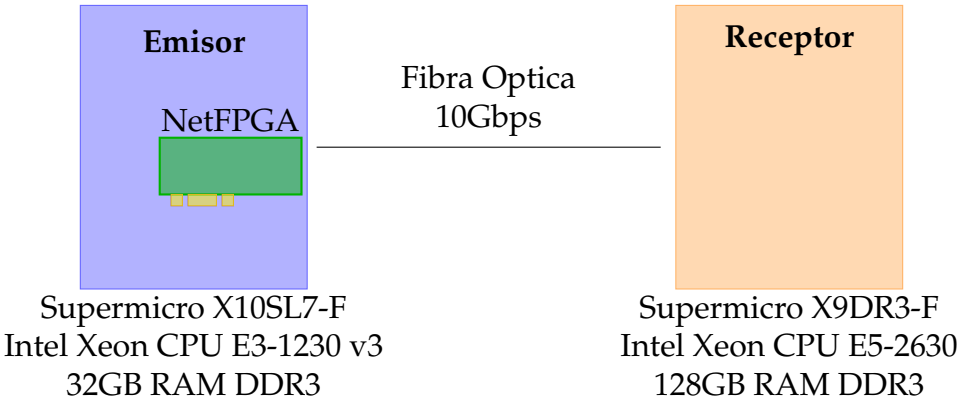

Figura 5.1: Arquitectura HW de las pruebas de rendimiento

El receptor tiene una Network Interface Card (NIC) de 10GbE de Intel con el chip 82599 conectada a una ranura PCI express asignada al primer nodo NUMA y utiliza HPCAP como driver. Como elemento de almacenamiento para guardar las capturas de tráfico que realiza DetectPro, se utiliza un RAID 0 de 10 discos mecánicos con una controladora hardware LSI Logic MegaRAID SAS 2208 que se encuentra conectado al segundo nodo NUMA. Por tanto, la distribución óptima de los procesos es dejar al hilo de captura de HPCAP en el nodo NUMA 0 y los procesos de DetectPro en el otro nodo NUMA, como se muestra en la tabla 2 de [19] y se reproducen en la tabla 5.8.

Afinidad fijada en el planificador de cores											
Nodo NUMA 0						Nodo NUMA 1					
0	1	2	3	4	5	6	7	8	9	10	11
C	-	-	-	-	-	P	E	D	-	-	-

P: hilo de proceso. E: hilo exportador de flujos.
C: hilo de captura. D: hilo de volcado de trazas.

Tabla 5.8: Afinidad de los procesos

Para configurar el RAID 0 se utilizó la configuración óptima para discos mecánicos dada en [18]: Write-Back como política de caché, un tamaño de strip de 1MB, con la caché de discos habilitada y utilizando el sistema de ficheros xfs.

Al comenzar esta fase de pruebas, se observó un fenómeno de degradación del rendimiento del RAID a medida que se escribían diferentes ficheros. También se observó una clara periodicidad de dicho fenómeno, que mostraba un periodo de aproximadamente 32 ficheros. El fenómeno observado era similar al que se detalla en la sección 4.2 de [16]. Este fenómeno producía graves problemas de rendimiento, y a pesar de que las cachés amotigüaban parte del efecto, cuando el rendimiento del RAID se degradaba por debajo del gigabyte, se producían muchas pérdidas de paquetes ya que el sistema no era capaz de procesar tráfico a 10Gbps.

```
mount -t xfs /dev/sdb /mnt/raid
```

Código 5.1: Comando utilizado para montar en el receptor el RAID con las opciones por defecto.

Finalmente, se encontró que era un problema de configuración: el RAID debía montarse con las opciones `inode64` y `largeio` para que mantuviese una tasa más o menos constante y suficiente como para poder procesar el tráfico a 10Gbps.

```
mount -t xfs -o largeio,inode64 /dev/sdb /mnt/raid
```

Código 5.2: Comando con las opciones correctas para mejorar el rendimiento.

En la figura 5.2 se muestra el resultado de una prueba realizada para comprobar el rendimiento del RAID. La prueba consistió en escribir 100 ficheros de 2GB con el script de bash mostrado en el 5.3. Se escogieron 100 ficheros para que se pudiese claramente apreciar el fenómeno, y 2GB es el tamaño de las trazas que escribe DetecPro. Así se pretende emular DetectPro escribiendo ficheros a máxima velocidad en el RAID y medir el rendimiento del mismo.

Como se puede apreciar en la figura, despreciando el efecto de las cachés en el primer fichero, el rendimiento del RAID se reduce de manera continuada hasta alcanzar un $55\,\%$ del rendimiento máximo cuando se monta con las opciones por defecto. Por otro lado, cuando se monta con los parámetros adecuados, muestra un rendimiento estable de entre 1.5 GBps y 1.6 GBps.

```
for i in {1..100}; do \
        taskset -c 7 dd if=/dev/zero of="/mnt/raid/${i}" \
        bs=1M count=2k oflag=direct ; \
done
```

Código 5.3: Prueba realizada para medir el rendimiento del RAID 0.

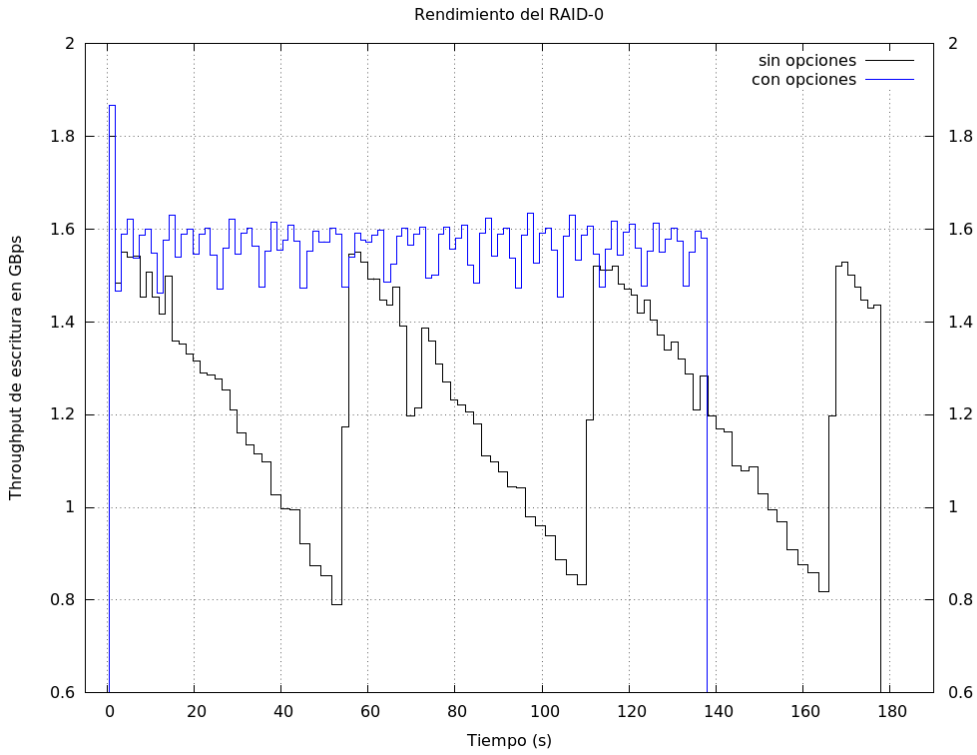

Figura 5.2: Resultado de las pruebas de rendimiento del RAID

5.4.2 Resultados y conclusiones

	Rendimiento (Gbps) $\bar{x} \pm \sigma$	Paquetes perdidos (%) $\bar{x} \pm \sigma$
DetectPro	9.652 ± 0.014	1.21 ± 0.09
DetectPro + ReTx	9.645 ± 0.013	1.25 ± 0.11

Tabla 5.9: Resultados de la prueba de rendimiento

Tras encontrar la configuración óptima, se tomaron los datos finales que se muestran en la tabla 5.9. En ella podemos observar que el rendimiento de DetectPro con detección de retransmisiones con el algoritmo 1 es algo menor que la versión sin modificar, pero prácticamente el mismo.

Cabe destacar que las medidas de rendimiento se han tomado sobre el tamaño del paquete ethernet, sin contar el preámbulo de la capa física, el "interframe gap" ni el CRC, por eso se obtienen tasas ligeramente inferiores a 10Gbps.

Por tanto, concluyo que esta prueba fue exitosa, y demuestra que es viable utilizar el algoritmo 1 para detectar retransmisiones en redes a 10Gbps.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

En este trabajo se han evaluado dos algoritmos de detección de retransmisiones en flujos TCP. Mientras que el algoritmo 2 ha demostrado no ser capaz de determinar de manera fiable si un flujo tiene suficientes retransmisiones, el algoritmo 1 ha resultado ser capaz de distinguir con certeza los flujos que tenían retransmisiones de los que no.

Ambos algoritmos cumplen los requisitos de memoria (solo requieren almacenar dos enteros por sentido de un flujo) y el algoritmo 1 ha probado que no se aprecia que impacte negativamente el rendimiento de DetectPro.

Se ha implementado un prototipo para probar ambos algoritmos que ha permitido estudiar el diseño de DetectPro, una aplicación bastante más compleja, lo que ha facilitado la labor de integración del algoritmo 1 al mismo.

6.2 Trabajo futuro

A continuación se comentan algunas líneas de trabajo futuro:

- Se debería estudiar si con alguna modificación el algoritmo 2 podría reducir sus tasas de error.
- También sería interesante estudiar si los FP del algoritmo 1 se pueden solucionar con alguna modificación.
- Ha faltado hacer pruebas con tráfico más variado, ya que se ha probado con trazas donde predominaba el tráfico web. Puede ser que con otras clases de tráfico o tráfico más variado se observen otras tasas de error.
- De cara a una posible publicación, sería interesante validar los resultados con tshark, por ser la herramienta de referencia de análisis de tráfico.

BIBLIOGRAFÍA

[1] https://www.wireshark.org Visitado el 2016-06-09.

[2] https://www.wireshark.org/docs/man-pages/tshark.html Visitado el 2016-06-09.

[3] https://github.com/torvalds/linux/blob/master/include/linux/list.h, Visitado 2016-06-03.

[4] Documentación de la libpcap. http://www.tcpdump.org Visitado el 2016-05-22.

[5] R. Braden. Requirements for Internet Hosts - Communication Layers. Technical Report 1122, 1989. http://www.ietf.org/rfc/rfc1122.txt.

[6] N. Brown. Linux kernel design patterns - part 2. https://lwn.net/Articles/336255/, Visitado el 2016-05-22.

[7] J. F. Kurose and K. W. Ross. *Computer Networking, A Top-Down Approach*. Pearson Addison-Weasley, 6th edition, 2010.

[8] K. R. Fall and W. R. Stevens. *TCP/IP Illustrated (Vol. 1): The Protocols 2nd Ed*. Addison-Wesley Professional Computing Series, 2012. Chap 13.

[9] V. Jacobson, B. Braden, and D. Borman. Tcp extensions for high performance. RFC 1323, RFC Editor, May 1992. http://www.rfc-editor.org/rfc/rfc1323.txt.

[10] B. Jenkins. The hash. http://burtleburtle.net/bob/hash/doobs.html, Visitado el 2016-05-22.

[11] B. Kim, D. Kim, and J. Lee. Lost retransmission detection for tcp sack. *IEEE Communications Letters*, 8(9):600–602, Sept 2004.

[12] K.-C. Leung, V. O.K. Li, and D. Yang. An overview of packet reordering in transmission control protocol (tcp): Problems, solutions, and challenges. *IEEE Transactions on Parallel and Distributed Systems*, April 2007. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4118693.

[13] R. Ludwig and R. H. Katz. The eifel algorithm: Making tcp robust against spurious retransmissions. *SIGCOMM Comput. Commun. Rev.*, 30(1):30–36, Jan. 2000.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgment options. RFC 2018, RFC Editor, October 1996. https://tools.ietf.org/html/rfc2018.

[15] M. Mellia, A. Carpani, and R. Lo Cigno. *TStat: TCP STatistic and Analysis Tool*, pages 145–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. http://dx.doi.org/10.1007/3-540-36480-3_11.

[16] V. Moreno. Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks. Master's thesis, Universidad Autónoma de Madrid, September 2012. http://hdl.handle.net/10486/12766.

[17] V. Moreno. *Harnessing low-level tuning in modern architectures for high-performance network monitoring in physical and virtual platforms*. PhD thesis, Escuela Politécnica Superior UAM, 2015. http://arantxa.ii.uam.es/~vmoreno/Publications/morenoPhD2015.pdf.

[18] V. Moreno, J. Ramos, , J. Garcia-Dorado, I. Gonzalez, F. Gomez-Arribas, and J. Aracil. Testing the capacity of off-the-self systems to store 10GbE traffic. *IEEE Communications Magazine*, 2015. http://arantxa.ii.uam.es/~vmoreno/Publications/Journals/morenoCOMMAGNT2015.pdf, p122.

[19] V. Moreno, P. M. Santiago del Río, J. Ramos, D. Muelas, J. L. García-Dorado, F. J. Gomez-Arribas, and J. Aracil. Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems. *International Journal of Network Management*, 24(4), 2014.

[20] J. Postel. Internet protocol. RDF 791, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc791.txt.

[21] J. Postel. Transmission control protocol. RDF 793, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[22] P. Roquero, J. Ramos, V. Moreno, I. González, and J. Aracil. High-speed tcp flow record extraction using gpus. *The Journal of Supercomputing*, 71(10):3851–3876, 2015.

[23] W. R. Stevens. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, RFC Editor, January 1997. http://www.rfc-editor.org/rfc/rfc2001.txt.

[24] I. Ucar, D. Morato, E. Magaña, and M. Izal. Duplicate detection methodology for ip network traffic analysis. (1311.4168v1), May 2013. http://arxiv.org/pdf/1311.4168.pdf.

[25] F. Vacirca, T. Ziegler, and E. Hasenleithner. An algorithm to detect {TCP} spurious timeouts and its application to operational umts/gprs networks. *Computer Networks*, 50(16):2981 – 3001, 2006.

[26] J. F. Zazo Rollón. Sistema basado en fpga para la captura de tráfico en redes multigigabit ethernet. 2014. http://hdl.handle.net/10486/662528.