UNIVERSIDAD AUTÓNOMA DE MADRID

Escuela Politécnica Superior

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

MONITORIZACIÓN, CAPTURA Y ALMACENAMIENTO INTELIGENTE DE TRÁFICO DE RED A 40GBPS

Guillermo Julián Moreno

Tutor: Francisco Gómez Arribas

Mayo 2016

Monitorización, captura y almacenamiento inteligente de tráfico de red a 40Gbps

Autor: Guillermo Julián Moreno
Tutor: Francisco Gómez Arribas

HPCN
Tecnología Electrónica y de las comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Mayo 2016

Resumen

Resumen El avance de las redes de datos lleva consigo la necesidad de monitorizarlas para detectar problemas, anomalías e intrusiones. Las velocidades cada vez más altas de estas tecnologías Ethernet hacen necesaria la aparición de soluciones específicas capaces de capturar ese tráfico, preferiblemente usando servidores estándar y hardware no especializado.

A lo largo de este trabajo se desarrollará una arquitectura para recibir y almacenar tráfico a 40 Gbps en tarjetas Intel y Mellanox usando como base el *driver* HPCAP, actualmente en explotación comercial para monitorización de redes 10 GbE. Esta arquitectura usará múltiples hilos de recepción para alcanzar una tasa suficientemente alta y aprovechará las características específicas de sistemas multiprocesador.

Previamente se estudiarán las soluciones existentes para monitorización para identificar las necesidades existentes, y se entenderá cómo funciona la pila de red del sistema Linux y la arquitectura de los *drivers* de los fabricantes de las tarjetas. Después, se planteará la arquitectura multihilo, que deberá usar primitivas de sincronización rápidas para evitar perder el rendimiento; y se integrará esta solución dentro de los *drivers* existentes de las tarjetas de Intel y Mellanox. Por último, se desarrollarán características adicionales para reducir la cantidad de datos recibidos y hacerlos manejables por aplicaciones de análisis o de almacenamiento a disco.

También se creará un entorno experimental en el que se pueda comprobar el rendimiento y correcto funcionamiento del desarrollo. Gracias a este entorno, se demostrará que la modificación de HPCAP es capaz de recibir a 40 Gbps para paquetes con tamaño suficientemente grande y que las funciones de reducción de datos son efectivas y mejoran el rendimiento.

Palabras clave Driver de red, Altas prestaciones, Captura de tráfico 40Gbps

ABSTRACT

Abstract The advance of data networks comes with the need for monitoring and detection of problems, anomalies and intrusions. The high speeds of the Ethernet technologies used incite the developments of specific solutions capable of capturing that traffic, preferably using standard servers and commodity hardware.

In this project, a new architecture for receiving and storing traffic at 40 Gbps line rate with Intel and Mellanox network cards will be developed, using as a starting point the HPCAP driver. This driver is currently in commercial operation for 10 GbE network monitoring. The architecture will use several reception threads to reach a high enough traffic rate, and will employ the specific features of multiprocessor systems to improve its performance.

First, the existent solutions for monitoring will be reviewed identifying existing needs, and the inner workings of the Linux's network stack and of the network drivers will be studied and understood. With this base, the multithreaded architecture will be designed using fast synchronization techniques to avoid performance hits, and this solution will be integrated with the existing Mellanox and Intel drivers. Then, additional features will be developed to reduce the amount of received data, in order to make it manageable by analysis or storage applications.

Additionally, an experimental environment will be developed to allow the testing and measurements of this project. Using this environment, we will prove that this modified version of HPCAP is capable of receiving traffic at 40 Gbps for frames of enough size, and that the features for data reduction are effective and improve the performance of the system.

Keywords Network drivers, High Performance, 40 Gbps traffic capture

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

GLOSARIO

API *Application Programming Interface*. 7, 8, 15, 33

BPF *Berkeley Packet Filter*, [10]. 20, 34

CPD Centro de Proceso de Datos. 1

DMA *Direct Memory Accesss*, un sistema que permite a los periféricos acceder directamente a la memoria del sistema. 12, 13

Driver Controlador de dispositivo: un programa que permite al sistema operativo interactuar con un periférico hardware. 1, 2, 5, 6, 8, 11, 12, 18

IRQ Petición de interrupción. 5

NAPI "New API", una API de Linux desarrollada para mitigar interrupciones en *drivers* de red y mejorar el rendimiento bajo condiciones de alta carga. 5, 6, 12

NIC Tarjeta de Interfaz de Red. 5, 7, 11, 12

NUMA *Non Uniform Memory Architecture*, diseño de sistemas multiprocesador con memoria local para cada procesador.. 25

Padding Datos "de relleno" que se añaden a una estructura de datos para alinearla a un tamaño concreto. 17, 18, 17, 18, 19

RAID *Redundant Array of Independent Disks*, una tecnología de virtualización de almacenamiento de datos que combina múltiples discos en una única unidad virtual, mejorando rendimiento y/o redundancia. 17

RoCE *RDMA over Converged Ethernet*, tecnología de Mellanox para permitir acceso directo a memoria remota (RDMA) a través de redes Ethernet. 13

RSS *Receive Side Scaling*, una tecnología para *drivers* de red que permite distribuir de forma eficiente los paquetes recibidos entre varias CPUs. 12, 13

1 INTRODUCCIÓN Y MOTIVACIÓN

En este capítulo se detallan las necesidades que dan lugar a este Trabajo Fin de Grado, planteando los objetivos que habrán de cumplirse para llevarlo a cabo. Además, se expondrá la estructura que seguirá este documento.

1.1 Motivación

Durante los últimos años, las necesidades de ancho de banda de Internet se han ido multiplicando a una velocidad difícil de imaginar en su momento. Muchos CPDs ya han desplegado redes de 10 Gbps, y los dispositivos capaces de funcionar a 40 Gbps o incluso a 100 Gbps ya empiezan a estar disponibles comercialmente.

Junto con estas redes de alta velocidad, aparece la necesidad de monitorizarlas. La información que se obtenga de la red puede guardarse para, posteriormente, diagnosticar problemas y evitar una infrautilización de los recursos o problemas en las aplicaciones que funcionen sobre la red. También es posible analizar esos datos en el momento de su recepción para detectar rápidamente anomalías, como puedan ser caídas de sistemas o accesos no autorizados.

Las soluciones de monitorización deben ser capaces por lo tanto de guardar la máxima información posible sin introducir distorsiones, y eso incluye no sólo la recepción de todos los paquetes que pasen por la red, sino también el marcado preciso de su tiempo de llegada.

En muchos casos se emplean sistemas o tarjetas de red diseñadas específicamente para entornos de monitorización. Es, sin embargo, un enfoque poco flexible, complejo y de alto coste. Así, se buscan soluciones de monitorización software, más flexibles y baratas, que funcionen sobre sistemas y tarjetas estándar.

El problema que se plantea es que los sistemas operativos modernos no están totalmente preparados para el manejo de estas velocidades. Incluso para 10 Gbps se hacen necesarias configuraciones específicas [9] que permitan al sistema

alcanzar las tasas que ofrece el hardware. Esto hace necesario el desarrollo a un nivel más bajo, modificando el funcionamiento de los *driver*s que controlan las tarjetas para obtener el rendimiento deseado.

Ahora bien, incluso a bajo nivel aparecen dos problemas que en muchas soluciones de monitorización se ignoran: el de marcado de paquetes o *timestamping* y el del orden de captura. A 40 Gbps, el tiempo entre paquetes puede llegar a ser de 16 nanosegundos[1], lo que da muy poco margen para realizar operaciones entre paquete y paquete.

Por otra parte, una recepción desordenada o imprecisiones mínimas en las marcas de tiempo pueden ser decisivas para que una herramienta de análisis detecte o ignore anomalías en el tráfico; o para que dé falsos positivos a raíz de fallos en la captura. Teniendo en cuenta que estos problemas ya los introducen las propias redes y sistemas a monitorizar, es necesario reducirlos todo lo posible en el lado de la captura. Estas necesidades llevarán al desarrollo a lo largo de este Trabajo Fin de Grado de una nueva arquitectura de recepción de tráfico, compatible con tarjetas de red estándar de Intel y Mellanox, y que sirva como punto de partida para sistemas de recepción no sólo a 40 Gbps sino también a tasas superiores.

Otro problema adicional que se plantea es el almacenamiento del tráfico recibido. No todas las aplicaciones son capaces o tienen la necesidad de realizar análisis de recepción en línea: puede ser interesante almacenar el tráfico en disco para realizar análisis *a posteriori* o sólo sobre intervalos de tiempo en los que se hayan detectado problemas en la red.

Sin embargo, a una tasa de 40 Gbps se necesita una gran cantidad de almacenamiento para todos los datos generados y que además sea suficientemente rápido para no ralentizar la captura. Los sistemas que cumplen estos requisitos son costosos y difíciles de montar, así que es necesario que la solución planteada de recepción sea capaz de sortear estas restricciones y reducir la cantidad de datos a almacenar sin perder información para el análisis posterior.

1.2 Objetivos

El objetivo principal que se plantea para este Trabajo Fin de Grado es la monitorización de tráfico en redes 40 GbE. Para ello, se ampliará el sistema de captura HPCAP [12], desarrollado en el grupo de investigación HPCN (Escuela Politécnica Superior, UAM) por Víctor Moreno a lo largo de su tesis doctoral.

HPCAP es un sistema de captura capaz de recibir y almacenar tráfico de red a 10 Gbps, desarrollado sobre el *driver ixgbe* de Intel para sus tarjetas de red 10 GbE. Previamente al desarrollo de este Trabajo Fin de Grado, el mantenimiento de este *driver* ha sido mi responsabilidad. Algunas de las tareas realizadas han sido mejorar la compatibilidad en diferentes distribuciones Linux, desarrollar el soporte para recepción de paquetes con etiquetas VLAN estándar y propietarias (Cisco), o facilitar el uso mixto de tarjetas con varias interfaces manteniendo unas

[1] Tomando paquetes de tamaño mínimo, 64 bytes, y añadiendo el *interframe gap* de 8 bits y la cabecera física.

en modo de captura de alto rendimiento y otras como interfaces de red Linux normales. Además, se han ampliado las capacidades del *driver* para que sea capaz de capturar no sólo en una sino en varias interfaces a altas tasas de tráfico, usando *buffers* intermedios más grandes con las *hugepages* de Linux.

Este trabajo previo sobre HPCAP permite plantear la ampliación del sistema para captura a tasas todavía mayores de tráfico, que será el propósito de este Trabajo Fin de Grado. Los objetivos para llevarlo a cabo serán los siguientes:

- Estudiar los límites de la arquitectura ya existente de HPCAP e identificar los puntos de mejora.
- Plantear una solución que permita la captura de tráfico a tasa de línea (40 Gbps) junto con marcas de tiempo precisas.
- Desarrollar métodos que reduzcan la cantidad de datos a guardar y hagan posible el almacenamiento de información sobre el tráfico que pueda ser analizada posteriormente.
- Verificar la portabilidad de la arquitectura, comprobando el funcionamiento correcto en tarjetas de Intel y Mellanox.
- Crear un entorno de pruebas que permita verificar y probar el rendimiento de sistemas de captura de tráfico de alta velocidad.

Dado el amplio ámbito de estos objetivos, el desarrollo se realizará de manera modular, cumpliendo los objetivos poco a poco y validando los resultados obtenidos a cada paso.

1.3 Estructura del documento

En el siguiente capítulo se revisará el estado del arte, estudiando la arquitectura de los *drivers* de red y sus cuellos de botella y detallando las soluciones ya existentes para salvar esos puntos de mejora.

Una vez estudiadas las soluciones existentes, a lo largo del capítulo 3 se planteará el sistema diseñado para lograr la recepción a 40 Gbps y su integración con los respectivos *drivers* base de las tarjetas de red.

Ese diseño se validará experimentalmente en el capítulo 4, donde se mostrarán los resultados de las pruebas y el rendimiento del sistema desarrollado.

Por último, en el capítulo 5 se expondrán las conclusiones del trabajo y se discutirán las líneas de trabajo futuro que surgen a partir del proyecto realizado.

2 ESTADO DEL ARTE

A lo largo de este capítulo se estudiarán las soluciones actuales de captura de tráfico a alta velocidad, empezando antes con una introducción al funcionamiento de *drivers* de red que permita identificar los cuellos de botella que los sistemas de monitorización deberán evitar.

2.1 Funcionamiento y cuellos de botella de un *driver* de red

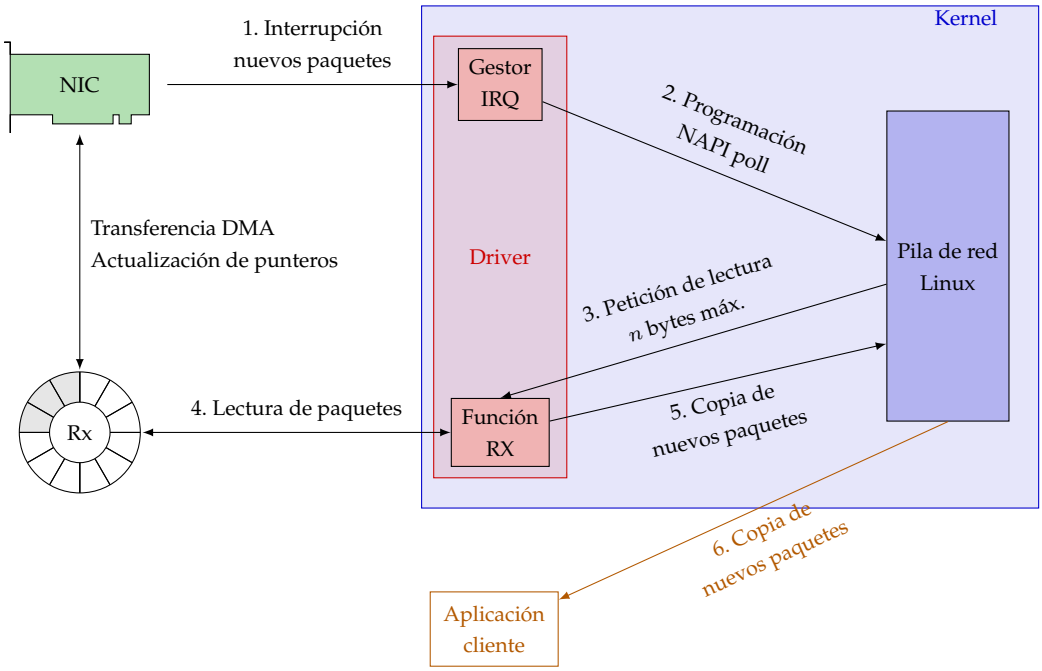

Figura 2.1: Esquema del funcionamiento de un *driver* de red en Linux.

A grandes rasgos, el funcionamiento de un *driver* de red en Linux es el que aparece en la figura 2.1. Cuando la NIC recibe nuevos paquetes, emite una IRQ que es recibida por la función correspondiente configurada por el *driver*. Éste se comunica con la pila de red del *kernel* de Linux, más concretamente

con el subsistema NAPI [1], avisando de la disponibilidad de nuevos paquetes. Además, desactivará las interrupciones de la tarjeta hasta que no se hayan leído todos los paquetes pendientes, para mejorar así el rendimiento.

El subsistema NAPI programará una llamada a la función de recepción del *driver* en función de la carga del sistema, pidiendo la lectura de hasta un cierto máximo de bytes. Cuando esa llamada se realice, el *driver* leerá de una región de memoria los paquetes que la tarjeta haya copiado a través de transferencias DMA, y los transferirá a la pila de red de Linux, que a su vez los gestionará y distribuirá a las aplicaciones cliente correspondientes.

Esta arquitectura es especialmente efectiva para sistemas de propósito general y muy flexible. Para tasas bajas, el uso de interrupciones como aviso de nuevos paquetes permite al sistema utilizar sólo los recursos necesarios para atender esos paquetes, dejando más tiempo de ejecución al resto de tareas del sistema.

Cuando la tasa aumenta, el subsistema NAPI evita los efectos perjudiciales que tendría un número excesivo de interrupciones: en lugar de realizar cambios excesivos de contexto gastando tiempo en atender a todas las interrupciones generadas, el *driver* las desactiva y simplemente lee los paquetes que la tarjeta va transfiriendo a memoria principal.

Ahora bien, como sistema de captura de alto rendimiento esta arquitectura no es del todo efectiva. Por un lado, se introduce una cierta latencia al tener que esperar a la primera interrupción y a la llamada correspondiente del subsistema NAPI para empezar a leer los paquetes del anillo de recepción de la tarjeta. Para tasas de tráfico suficientemente altas, el tiempo invertido en atender a esa interrupción y el que tarde el sistema Linux en llamar a la rutina de recepción puede ser demasiado alto y provocar la pérdida de los primeros paquetes a recibir. Además, NAPI limita la cantidad de paquetes a recibir en cada llamada, por lo que las interrupciones se están invocando continuamente a pesar de que no aporten información: con una tasa suficientemente alta, el anillo de recepción siempre tiene paquetes.

Por otra parte, con este método se realizan dos copias redundantes: del anillo de recepción a la pila de red de Linux y de ésta a la aplicación cliente. Dado que los accesos a memoria principal son costosos, es deseable reducirlos al máximo para obtener un mejor rendimiento.

2.2 Sistemas de captura de tráfico

Para el sistema operativo Linux se pueden encontrar multitud de sistemas de captura de tráfico. Entre los más conocidos se puede encontrar *libpcap* y sistemas basados en él, como *tcpdump* o *Wireshark*. Ahora bien, tal y como se comentaba en la sección anterior, estos sistemas no están preparados para la captura a altas tasas de tráfico y surgen alternativas más eficientes y con mejor rendimiento. A lo largo de esta sección revisaremos algunos de ellos.

2.2.1 Linux PACKET_MMAP

Una opción para captura de alto rendimiento integrada en el propio *kernel* de Linux es la configuración llamada PACKET_MMAP [3]. Esta opción permite acceder directamente a los anillos de recepción del sistema desde espacio de usuario, de tal forma que no hay que realizar llamadas a sistema ni hacer copias adicionales para recibir cada paquete.

Aunque es un método más eficiente que la captura paquete a paquete, se sigue usando la pila de red de Linux, lo que hace difícil la mejora y modificación para obtener mayor rendimiento o poder usar los marcados de tiempo *hardware* que ofrezca la tarjeta.

2.2.2 PF_RING ZC

PF_RING ZC (*Zero Copy*) es la versión modificada de PF_RING DNA [15]. Se trata de un sistema comercial de captura de tráfico a 10 Gbps preparado para procesar y reenviar paquetes copiando sólo los datos de la NIC a la memoria de sistema, saltándose la pila de red del sistema. Una de sus principales fortalezas es la virtualización: PF_RING ZC permite a máquinas virtuales sobre Linux acceder al tráfico a 10 Gbps, e incluso realizar operaciones sin copias de memoria entre esas máquinas virtuales para mejorar el rendimiento.

Sin embargo, todavía no ofrece captura a 40 Gbps, marcado de tiempo preciso o soporte para NICs distintas de las de Intel.

2.2.3 Netmap

Netmap [14] es un sistema de generación y recepción de tráfico orientado a entornos de alto rendimiento, capaz de funcionar con enlaces de 40 Gbps con tarjetas Chelsio. Para ello usan un *driver* modificado capaz de acceder directamente a los *buffers* de la tarjeta y que agrupa las operaciones por lotes para mejorar el rendimiento, y usan múltiples colas y anillos para paralelizar la carga de trabajo.

A nivel de usuario, Netmap expone descriptores de fichero que se pueden operar con las llamadas estándar de Linux (`ioctl` y `poll`). Ahora bien, es un sistema que no se enfoca especialmente al almacenamiento de los paquetes y que no pone marcas de tiempo en la recepción.

2.2.4 DPDK

Desarrollado por — entre otros — Intel, DPDK (*Data Plane Development Kit*) [2] es un conjunto de librerías y *drivers* para el envío y procesado de paquetes a alta velocidad. Soporta tarjetas de múltiples fabricantes, incluyendo Mellanox e Intel; y permite captura a 40 Gbps con NICs apropiadas.

DPDK expone una API que permite a las aplicaciones configurar el sistema de captura y recibir tráfico. También se pueden usar herramientas ya desarrolladas, como MoonGen [5] que permite generar y recibir paquetes usando programas Lua interpretados en tiempo real.

La principal desventaja de DPDK es que obliga al programador a realizar cualquier tarea adicional a la captura en espacio de usuario, de tal forma que

resulta difícil aplicar marcas de tiempo precisas al no poder detectar el momento exacto en el que llega cada paquete.

2.2.5 HPCAP

HPCAP [12] es un sistema de captura desarrollado en el grupo HPCN de la Escuela Politécnica Superior (UAM) por Víctor Moreno, que permite la captura de tráfico a alta velocidad con marcado de tiempo preciso; y que además está optimizado para permitir el almacenamiento de ese tráfico en disco para su posterior análisis.

El sistema consiste en la modificación del *driver* original de Intel para tarjetas 10 GbE, de tal forma que los paquetes capturados no pasan por la pila de red del sistema sino que se guardan en un *buffer* intermedio en memoria del sistema. Las aplicaciones cliente podrán leerlos y procesarlos a través de la API que expone HPCAP, o realizar un guardado directo a disco sin copias adicionales.

Para mejorar el rendimiento del almacenamiento, el *driver* inserta unas cabeceras antes de copiar cada paquete al *buffer* intermedio, que marcan la longitud del paquete y su tiempo de llegada. De esta forma no hay estructuras adicionales que consultar, lo que permite el guardado directo a disco por bloques de 1 MB y evita tener que copiar paquete a paquete, que es menos eficiente.

La comunicación con las aplicaciones cliente de HPCAP se realiza a través de descriptores de fichero generados para cada interfaz. Una aplicación que quiera leer paquetes del *buffer* de HPCAP abrirá ese descriptor de fichero y, usando llamadas `ioctl`, se comunicará con el *driver* para recibir el número de bytes disponibles para su lectura en el *buffer* y enviar los bytes leídos.

Internamente, HPCAP mantiene un registro de la última posición leída por cada cliente y de la última posición escrita por el hilo de recepción de tráfico. De esta forma, en lugar de reservar y transmitir memoria por cada cliente, el *driver* copia los datos a un único *buffer* compartido por todos los clientes, escribiendo sólo cuando haya espacio disponible. Así se evitan copias adicionales que afectarían al rendimiento.

El *driver* de HPCAP es capaz de recibir paquetes de tamaño mínimo (64 bytes) a la tasa máxima del enlace sin pérdidas. Además, es la base de un sistema de monitorización y análisis de tráfico [13] actualmente en explotación comercial.

2.3 Conclusiones del capítulo

En el momento actual no existe un gran número de soluciones de captura de tráfico a 40 Gbps, y ninguna de ellas contempla especialmente las marcas de tiempo precisas, el orden de paquetes en la recepción o el almacenamiento del tráfico.

El sistema HPCAP que se usará como base en este Trabajo Fin de Grado tiene todas las características necesarias como arquitectura de recepción de alto rendimiento, evitando los cuellos de botella existentes en el sistema de red de Linux.

Sin embargo, todavía es necesario plantear una arquitectura capaz de recibir a una tasa mayor de 10 Gbps, que es el límite actual de HPCAP, manteniendo la compatibilidad con la API que da acceso a las aplicaciones cliente. Esta solución se desarrollará en el siguiente capítulo.

3 DESARROLLO E IMPLEMENTACIÓN

A lo largo de este capítulo se planteará el diseño de la arquitectura de recepción a 40 Gbps y las soluciones que permitan reducir datos y almacenar el tráfico recibido incluso a tasas demasiado altas para sistemas de almacenamiento comunes.

3.1 Arquitectura de los *drivers* originales

Para poder adaptar HPCAP a las NICs de 40 Gbps de Intel y Mellanox es necesario utilizar los *driver*s originales de los fabricantes y modificarlos para que sea el código de HPCAP el encargado de la recepción. Esto implica conocer su funcionamiento no sólo a grandes rasgos (sección 2.1), sino también las rutinas de comunicación con el hardware y de reserva de recursos de esos *driver*s originales.

A pesar de ser de fabricantes distintos, los dos *driver*s siguen el mismo proceso para inicializarse y para la recepción de paquetes. Cuando el *driver* se inserta en el sistema Linux, lee los parámetros de configuración de la línea de comandos y empieza a reservar recursos acorde a esos parámetros. Entre esos recursos están las regiones de memoria de los anillos de recepción, donde la NIC escribirá los paquetes recibidos; y estructuras en memoria compartida entre software y

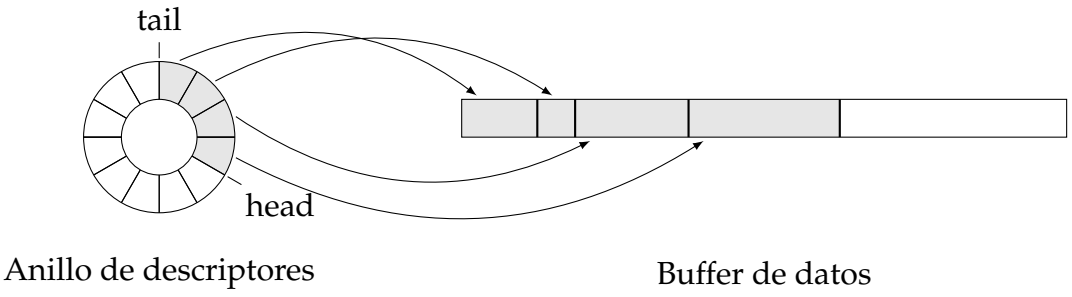

Figura 3.1: Esquema del anillo de descriptores para la recepción de paquetes. Cada descriptor tiene un puntero a los datos del paquete recibido y un campo que marca si el descriptor tiene o no datos. En gris, los datos y descriptores listos para ser leídos por el *driver*.

hardware, de donde se leerán datos como los descriptores de paquetes y los punteros de cabecera y cola del anillo.

El anillo de descriptores (figura 3.1) es la estructura de datos que permite la transferencia de paquetes desde la NIC al *driver*. Cada descriptor de paquete es una estructura de tamaño fijo que contiene una variable con el estado del descriptor, con los marcadores que determinan si tiene datos, si ha tenido algún error o si el paquete está dividido en más descriptores; y un puntero a los datos del paquete de red correspondiente, que han sido copiados a la memoria del sistema por la NIC.

Este anillo se comporta como una cola, aunque implementado de manera distinta para mejorar el rendimiento. El puntero de cabecera, *head*, que marca hasta dónde hay paquetes pendientes de leer, no aparece explícitamente y no se actualiza por la NIC al enviar nuevos paquetes al sistema. En su lugar, para comprobar si hay o no nuevos paquetes por leer, el *driver* simplemente lee el campo de estado del siguiente descriptor de paquete para saber si tiene datos o no. Esto supone un ahorro de rendimiento, ya que se ahorra una escritura adicional que sería especialmente lenta, al no poder agruparse con otras escrituras a disco como sí se puede hacer con la escritura de datos o descriptores.

La rutina de recepción, llamada por el subsistema NAPI, va leyendo ese anillo de descriptores y enviando los nuevos paquetes que encuentra al sistema operativo, que se encargará de repartirlos a su destino correspondiente.

Una vez que los paquetes se han leído, no sólo hay que actualizar el puntero de cola, sino también notificar al sistema DMA de que la memoria que usaban queda libre para su uso por la tarjeta. Realizar esta operación paquete por paquete introduce un coste adicional que se puede ahorrar acumulando varias operaciones que se realizan de una vez, aunque no deben acumularse demasiadas para evitar que la NIC se quede sin descriptores donde copiar nuevos paquetes y los descarte.

3.1.1 Integración con HPCAP, ajustes del hardware y configuración

Para que HPCAP funcione con los *driver*s descritos anteriormente, la principal modificación a hacer es sustituir la rutina de recepción original. Para ello, se evita el registro de la interfaz en el subsistema NAPI para evitar interferencias del sistema, se desactivan las interrupciones de la NIC, y se lanzan los hilos de recepción de HPCAP cuando el resto del *driver* se haya inicializado correctamente.

Los hilos de recepción de HPCAP consultan continuamente el anillo de la tarjeta, copiando los paquetes nuevos y liberando los descriptores ya leídos. Estos hilos no son interrumpidos en ningún momento por el *kernel* y se asignan a una CPU fija, evitando cambios de contexto o paradas que reduzcan el rendimiento del sistema.

Además de esa sustitución, se añaden a las rutinas de inicialización las funciones que reciben y analizan los argumentos pasados al *driver* para configurar HPCAP apropiadamente, y se aumenta el tamaño del anillo de descriptores para mejorar el rendimiento.

Para optimizar la tasa de recepción es necesario cambiar la configuración de la tarjeta con respecto a las opciones por defecto. Las tarjetas están preparadas para realizar varias operaciones en hardware, como la eliminación de cabeceras VLAN o el análisis del protocolo TCP (*TCP offloading*) para ahorrar trabajo a la CPU. También es estándar la tecnología RSS, que permite distribuir el trabajo de recepción de tráfico entre varios núcleos del procesador. Para ello, se crean varios anillos de recepción, cada uno atendido por un núcleo distinto. La tarjeta se encarga de distribuir los paquetes a cada uno de esos anillos, normalmente tratando de mantener paquetes de un mismo flujo dentro del mismo anillo.

Todas estas características no son necesarias si el objetivo es capturar todo el tráfico usando la menor cantidad de recursos posible. Esas operaciones adicionales bajan el rendimiento y es necesario desactivarlas. En ambas tarjetas se reduce el número de colas RSS a 1 y se desactiva la descarga de VLAN.

Para las tarjetas Mellanox, se realizan los siguientes cambios adicionales:

- Se desactiva la distribución RSS para paquetes UDP.
- Se configura la función más rápida (*XOR*) para la distribución RSS.
- Se desactiva el modo RoCE.
- Se activa el modo de RSS estático, que da mejor rendimiento a costa de menor flexibilidad en las reglas de distribución.

Para las tarjetas Intel, los cambios son los siguientes:

- Se desactiva el modo *packet-split* para que la tarjeta guarde cada paquete en un único *buffer*.
- Se desactiva la copia directa a *socket buffers* (estructuras internas del sistema de red de Linux) y se reserva una zona de memoria contigua que actúa de *buffer* DMA de la tarjeta.

Una vez hecha la integración del código existente de HPCAP con los *drivers* de Intel y de Mellanox, el siguiente es el diseño de una arquitectura que sea capaz de recibir tráfico a 40 Gbps.

3.2 Recepción a 40 Gbps

Operación	Ciclos	Nanosegundos	% Tiempo disponible
Acceso caché L1	4	1,18	7 %
Acceso caché L2	12	3,53	21 %
Acceso caché L3	44	12,94	77 %
Branch mispredict	16	4,71	28 %

Tabla 3.1: Latencias en CPUs Intel con arquitectura Skylake [6], comparando con el tiempo disponible entre paquetes de 64 bytes a tasa de 40 Gbps en un procesador a 3,40 Ghz.

Tal y como se comentaba en la introducción, la recepción a 40 Gbps es un desafío importante por las restricciones de tiempo. A esta tasa, los paquetes

pueden llegar cada 16 nanosegundos, lo que en un procesador de gama alta a 3,40 GHz apenas significan 57 ciclos.

Esta restricción de tiempo hace imposible que un único hilo pueda soportar toda la carga de tráfico. Un fallo de caché o del predictor de saltos puede consumir un cuarto del tiempo disponible para el procesado de un paquete, de tal forma que es prácticamente imposible procesar los paquetes a esa tasa incluso aunque no hubiese que copiarlos después a un *buffer* intermedio.

Así, se hace necesario el uso de varios hilos de lectura que sí puedan soportar conjuntamente estas altas tasas de tráfico. El problema a resolver será el de la concurrencia y sincronización entre hilos en tres puntos clave:

1. Lectura de paquetes del anillo de recepción.
2. Devolución ordenada de los descriptores leídos a la tarjeta.
3. Copia al *buffer* intermedio de HPCAP.

Por supuesto, todas las operaciones de sincronización tendrán que hacerse sin mecanismos de bloqueo, que son demasiado lentos para el procesado a tasa de 40 Gbps. A *priori*, la solución sería la implementación de una cola sin bloqueos (por ejemplo como la desarrollada en [8]). Sin embargo, esta situación específica y las características de los lectores y de los clientes que accederán al *buffer* intermedio de HPCAP permitirán la simplificación de las operaciones y la reducción de los mecanismos de concurrencia necesarios.

3.2.1 Lectura del anillo de recepción

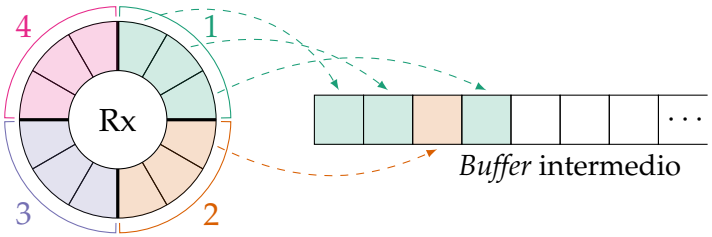

Figura 3.2: División del anillo de recepción en cuatro segmentos fijos para los hilos, mostrando el efecto que tendría en el *buffer* de escritura.

A la hora de la lectura, un problema a tener en cuenta es el de la caché. Un fallo de caché puede provocar una latencia de muchos ciclos de reloj (tabla 3.1), y con ello la pérdida de paquetes. Si todos los hilos de recepción tratan de leer todos los paquetes del anillo de recepción, los patrones de acceso no van a ser predecibles. Cuando un hilo acceda a un descriptor, el sistema cargaría en la caché de su núcleo descriptores cercanos para un acceso más rápido. Ahora bien, es posible que el resto de hilos consuman esos paquetes, de tal forma que el siguiente paquete disponible ya no estaría cargado en caché y habría una latencia adicional al acceder a él.

Para evitar este problema, a cada hilo de lectura se le asigna un segmento fijo del anillo de recepción como en la figura 3.2. De esta forma cada hilo siempre accederá a los mismos descriptores, lo que permitirá incluir instrucciones de

prefetch para garantizar en la medida de lo posible que los datos a acceder estén siempre en la caché.

Esta asignación fija de segmentos del anillo también evita problemas de concurrencia, ya que no hay competición por acceso a los mismos recursos.

Por último, se minimiza el desorden en la lectura. Si todos los hilos leyesen los mismos descriptores, lo más probable es que los paquetes se copien en un orden distinto al que tenían cuando llegaron a la tarjeta. Sin embargo, de esta forma los hilos copian secuencialmente la mayor parte de los paquetes, y sólo se produciría desorden cuando haya dos o más hilos con paquetes pendientes al mismo tiempo. Con tasas bajas de paquetes por segundo, esto apenas ocurrirá durante pequeños intervalos de tiempo.

3.2.2 Devolución ordenada de los descriptores leídos

El siguiente problema a resolver es el de la devolución ordenada de los descriptores ya leídos y de las regiones de memoria correspondiente a la tarjeta, para que ésta los pueda reutilizar para los nuevos paquetes que lleguen. Es importante que esta devolución se realice por lotes al ser más eficiente. Además, ha de ser secuencial: no se pueden dejar atrás descriptores sin devolver ya que pueden provocar fallos de segmentación o la tarjeta los puede sobrescribir antes de tiempo.

Al segmentar el anillo de recepción, esta tarea se simplifica considerablemente y no son necesarios mecanismos de sincronización complejos. Cada hilo puede liberar los descriptores de su segmento por lotes sin problemas: la única restricción es que ya se hayan liberado todos los descriptores del segmento anterior.

La solución implementada es la siguiente. A cada hilo se le asignan dos variables, `can_free` y `freed_last_rxd`. La primera variable indica si el hilo puede liberar sus descriptores, y la segunda si ya ha liberado el último de los descriptores de su segmento. A la hora de liberar los descriptores, se hace la primera comprobación sobre `can_free`: si es 1, se liberan sin problemas. Además, si se llega al último descriptor, `freed_last_rxd` se pone a 1 y `can_free` a 0 (hay que esperar, de nuevo, a que el segmento anterior libere sus descriptores).

Si, por otra parte, `can_free` es 0, se lee la variable `freed_last_rxd` del segmento previo. Si es 1, indica que ya se pueden liberar los descriptores del segmento actual, así que se reinicia su valor a 0 y `can_free` se pone a 1. Es necesario que esta comprobación se haga sólo cuando `can_free` sea 0. El pseudocódigo de este procedimiento se puede ver en el algoritmo 3.1.

La única técnica de concurrencia usada es que la variable `freed_last_rxd` ha de ser atómica, de tal forma que su valor está constantemente sincronizado entre todos los hilos y no hay posibilidad de que un hilo la lea mientras otro está escribiendo su valor. Además, para mantener un orden estricto en la lectura y escritura de las estructuras de datos.

```
function FREE_DESCRIPTORS(thread, descr)
    prev ← PREVIOUS_THREAD_OF(thread)
    if can_free[thread] == 0 && freed_last_rxd[prev] == 1 then
        freed_last_rxd[prev] ← 0
        can_free[thread] ← 1
        READ_BARRIER( )
    end if
    if can_free[thread] == 1 then
        RELEASE_TO_NIC(descr)
        if IS_LAST_IN_SEGMENT(descr, thread) then
            WRITE_BARRIER( )
            freed_last_rxd[prev] ← 1
            can_free[thread] ← 0
        end if
    end if
end function
```

Algoritmo 3.1: Algoritmo de liberación de descriptores

3.2.3 Copia al *buffer* intermedio de HPCAP

El último problema a resolver es el de la copia al *buffer* intermedio de HPCAP, de donde leerán después los clientes, manteniendo además la API y el funcionamiento ya existente descrito en la sección 2.2.5.

Para ello se empleará una variable atómica que marque la siguiente posición de escritura en el *buffer*. Cuando un hilo quiera escribir un nuevo paquete, aumentará esa variable en el tamaño del paquete más la cabecera correspondiente, reservando ese espacio a la vista del resto de hilos. Una vez hecha esa reserva, escribirá los datos y continuará con el siguiente paquete.

El siguiente paso es notificar a los clientes que hay nuevos datos disponibles para leer. De nuevo, se pueden aprovechar cuestiones de diseño de HPCAP para evitar mecanismos de bloqueo para gestionar la concurrencia. En una cola sin bloqueos habría que evitar de alguna forma que los clientes puedan leer datos que no se han terminado de escribir. Sin embargo, en HPCAP los clientes están en espacio de usuario, y eso induce un retardo que hace innecesarios más mecanismos de sincronización.

El peor caso ocurriría cuando un hilo actualiza la variable reservando espacio para un nuevo paquete, y justo en el ciclo siguiente se lee esa variable en otro hilo desde una llamada `ioctl` realizada por un cliente. Ahora bien, para que ese valor leído llegue a espacio de usuario primero ha de realizarse una copia a la memoria de espacio de usuario y se tiene que hacer un cambio del modo *kernel* a modo espacio de usuario para que la aplicación cliente retome el control de la ejecución, lo que habitualmente lleva más de 50 nanosegundos, unos 170 ciclos de reloj a 3,40 GHz. En ese tiempo se asegura que la copia se ha realizado correctamente, dado que los hilos de lectura no son interrumpidos en ningún momento. Así, incluso en el caso en el que menos tiempo pasaría desde la reserva del espacio de escritura hasta que la aplicación cliente lea esos datos queda garantizado que la copia se realiza correctamente.

Como optimización adicional, en un principio habría que asegurar que esta variable se mantuviese entre 0 y el tamaño del *buffer* de escritura, pero esto implicaría no una operación atómica (sumar un valor y leer el resultado) sino varias, ya que habría que calcular el módulo si el valor obtenido supera el tamaño del *buffer*. Esto implicaría más mecanismos de sincronización y por lo tanto un rendimiento menor.

Por suerte, se puede aprovechar el desbordamiento de enteros de C para hacer ese módulo implícitamente. Por el diseño de HPCAP, el tamaño del *buffer* intermedio es siempre una potencia de 2, luego se puede dejar que la variable atómica desborde normalmente y después calcular el módulo tamaño del *buffer* para obtener la posición real, ya que $(x \mod 2^{32}) \mod 2^n = x \mod 2^n$, donde 2^n es el tamaño del *buffer* con $n \leq 32$.

Detección de llenado del *buffer* Es necesario asegurarse de que, al copiar los paquetes, no se sobrescriben datos que todavía no hayan leído los clientes. Dado que las reservas de tamaño se hacen de forma atómica, no es posible comprobar si hay espacio y luego hacer la reserva: podría darse una condición de carrera en la que, habiendo espacio para un único paquete, dos hilos comprueben a la vez que tienen espacio suficiente y por lo tanto sobrescriban el *buffer*.

La solución es una limitación *a priori* de la cantidad de datos que puede copiar cada hilo. Antes de entrar en el bucle de lectura, cada hilo comprueba el espacio libre e en el *buffer* (distancia entre desplazamientos de escritura y lectura) y calcula el límite de datos a copiar como e/n, donde n es el número de hilos. De esta forma, la condición de carrera anterior desaparece: si no hay espacio para un paquete, al dividir ese espacio entre n hilos ninguno de ellos lo copiará al *buffer* y por lo tanto no habrá sobrescritura.

Además, esta solución no afecta al rendimiento: mientras el *buffer* no esté cerca de llenarse, el coste de salir del bucle de lectura y recalcular el espacio libre es mínimo en comparación con el tiempo invertido en leer y copiar todos los paquetes. Si por otra parte el *buffer* está cerca de llenarse, cada hilo acabaría llenando aproximadamente la misma n-ésima parte del espacio restante antes de comprobar que no hay más espacio y salir del bucle.

Padding Hay que tener en cuenta que el propósito de HPCAP es la captura y guardado en disco de los paquetes, y este es el principal caso de uso a optimizar.

Precisamente para facilitar esas copias de *buffer* a ficheros, HPCAP implementa un *padding* [11] de tal forma que la aplicación cliente siempre pueda copiar un número entero de paquetes en archivos con un tamaño exacto de 2 GB, sin dejar paquetes cortados entre fichero y fichero. Esto permite maximizar el rendimiento del RAID y del acceso a memoria (las lecturas y escrituras siempre están alineadas a tamaño de bloque) y usar ciertas opciones que mejoran el rendimiento.

En esta versión, por lo tanto, hay que seguir manteniendo el *padding* para que las copias a disco sigan siendo óptimas, aunque el método para insertarlo tiene que cambiar. La implementación original mantenía un desplazamiento dentro del fichero, que se actualizaba cada vez que se copiaba un paquete. En el momento

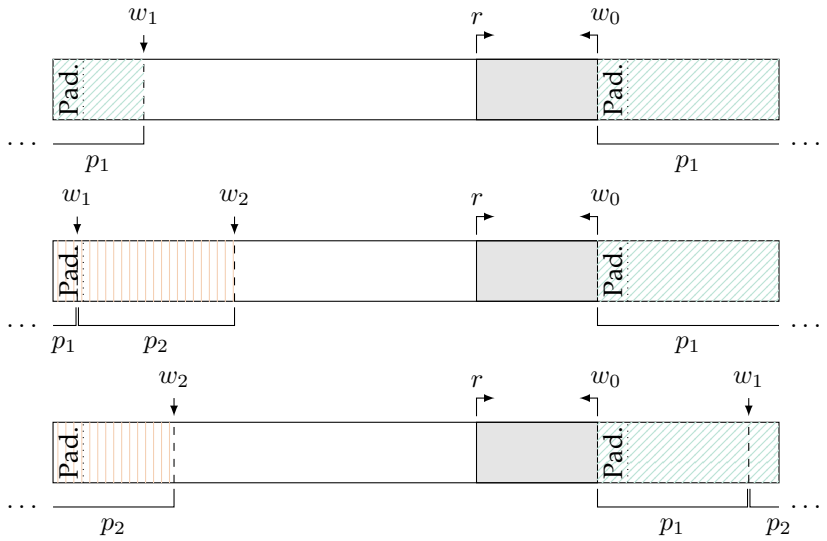

Figura 3.3: Esquema de implementación de *padding*. En el primer caso, el mismo hilo escribe el *padding* a principio y final de fichero. En el segundo y tercer no tiene espacio para escribirlo al principio, por lo que se completa al escribir otro paquete.

de la copia, comprobaba primero si el paquete a copiar entraba dentro del tamaño de fichero. Si no lo hacía, se incluía una cabecera marcando la longitud del *padding* reiniciando el desplazamiento dentro del fichero a cero, y a continuación copiando el paquete.

Tal y como se ha diseñado la escritura concurrente, es imposible implementar el *padding* de la misma forma, ya que siempre se reserva el tamaño del paquete tanto si cabe en el fichero actual como si no. La solución al problema será llenar ese espacio ya reservado con *padding*, teniendo en cuenta que es posible que no haya espacio para la cabecera completa al principio o al final del fichero. Una vez terminada esa escritura, se reservará de nuevo espacio para copiar el paquete que se iba a copiar en un principio y se continuará normalmente con el proceso.

Dado que el desplazamiento dentro del *buffer* se guarda sin hacer el módulo, no hace falta mantener otro contador separado para el desplazamiento dentro del fichero. Para obtenerlo, sólo es necesario calcular ese desplazamiento módulo el tamaño del fichero. La única precaución a tener es que el tamaño del fichero ha de ser divisor del máximo valor de la variable (2^{32} bytes, 4 GB).

Ese desplazamiento permitirá al *driver* detectar cuándo, una vez reservado el espacio en el *buffer*, el paquete a copiar no entra por completo en el fichero y es necesario insertar un *padding*. El primer paso es comprobar si en el espacio restante del fichero entra el paquete que está pendiente de escribir y además otra cabecera adicional. Es necesario tener en cuenta la inclusión de esa cabecera adicional para evitar casos en los que no quede espacio siquiera en el fichero para una cabecera completa que marque el *padding*.

Si no hay espacio suficiente, se introduce una cabecera con los marcadores de tiempo a 0 y que indique la longitud del *padding*, en este caso, hasta el final de fichero. Además, se comprueba si desde el principio del fichero hasta la última posición reservada (w_1 en la figura 3.3) hay espacio suficiente para incluir otra

cabecera de *padding*. Si lo hay, ese mismo hilo escribirá la cabecera y después se continuará la escritura de paquetes con normalidad.

Si por el contrario no se puede incluir la cabecera a comienzo de fichero, la siguiente escritura (que puede ser realizada por ese mismo hilo o por otro cualquiera, es indiferente) se encargará de hacerlo. Para ello, se comprueba si la diferencia entre el inicio del fichero y la primera posición en la que hay que escribir (w_1) es menor que la longitud de cabecera, en cuyo caso se introduce ese *padding* a comienzo de fichero, que llegará hasta la última posición que se había reservado (w_2 en la figura 3.3).

3.2.4 Marcas de tiempo

El último paso en la recepción de los paquetes es el marcado de su tiempo de llegada. En este caso, las tarjetas Mellanox tienen una ventaja: son capaces de marcar los tiempos de llegada en el hardware.

Para obtener las marcas de tiempo, durante la configuración inicial de la tarjeta se activa el filtrado hardware. En la recepción, cada descriptor incluirá un marcado de tiempo usando un reloj interno de la tarjeta, que después habrá que convertir a la hora local y guardar en la cabecera del paquete en un formato legible con toda la precisión posible, que en el caso de la tarjeta de Mellanox implica almacenarlo en nanosegundos.

En el caso de las tarjetas de Intel, no hay marcado hardware y éste se tiene que realizar en los hilos de recepción. Para ello, se usa la función `getnstimeofday()`, ya evaluada como la más precisa para las necesidades de HPCAP [12]. A pesar de ello, no podrá llegar a ser tan precisa como el marcado hardware, ya que en este caso entran en juego varios factores que introducen imprecisiones en las marcas de tiempo software

Por un lado, las marcas son susceptibles al rendimiento de los hilos: cuanto más tarden en procesar cada paquete desde que éste llega al anillo de recepción, más desplazado estará el tiempo marcado con respecto al de llegada real. Además, dado que la tarjeta Intel trabaja por lotes de paquetes (8 ó 4 descriptores cada vez según la configuración [4]), puede ocurrir que paquetes con tiempos distintos de llegada se marquen de manera muy seguida por el hilo de recepción correspondiente, distorsionando así los datos del tráfico.

3.3 Almacenamiento del tráfico: filtrado y guardado selectivo

En la sección anterior se ha expuesto un diseño para la recepción de tráfico a 40 Gbps, pero falta considerar las opciones disponibles para poder almacenar todo ese tráfico para su posterior análisis.

La opción directa, el almacenamiento de todo el tráfico, no es factible si se quieren mantener unos costes bajos. Como se puede ver en la tabla 3.2, incluso para un entorno de captura conservador (24 horas de tráfico almacenado a 20 Gbps) los costes son excesivamente altos. La única opción factible serían discos

Tipo	Modelo	Velocidad (MB/s)	Capacidad (TB)	Discos	Coste €
SSD	Samsung 850 Pro	520	2	106	86,496 €
	Intel DC 3700	970	1.6	132	158,400 €
Mecánico	HGST Ultrastar	220	6	36	21,600 €
	Seagate Entrp.	240	4	53	26,500 €

Tabla 3.2: Estimaciones de coste de los discos necesarios para un sistema de captura de 24 horas de tráfico a 20 Gbps.

mecánicos, y aún faltaría tener en cuenta el coste del sistema RAID que soporte como mínimo 36 discos.

Se hace necesario por lo tanto desarrollar sistemas que permitan filtrar el tráfico recibido o guardar sólo la información estrictamente necesaria de cada paquete. A lo largo de esta sección se planteará el diseño de estas soluciones para el filtrado.

3.3.1 Filtrado de paquetes

Dado que el filtrado ha de hacerse en el bucle de recepción, es necesario que el sistema sea simple y se puedan evaluar los filtros rápidamente. Para ello, se traslada toda la complejidad al espacio de usuario, donde se traducirán los filtros deseados a una estructura de datos que simplifique el análisis.

Esta estructura será una lista de cadenas y su correspondiente posición de inicio en el paquete. Un elemento de esa lista podría ser, por ejemplo, la cadena que contenga la dirección MAC `AA:BB:CC:DD:EE:FF` y la posición de inicio 0, de tal forma que el filtro aceptaría sólo paquetes con esa MAC destino.

Además, se podrá configurar el filtrado para que sólo se acepte un paquete si todas las cadenas coinciden, o configurar cadenas con coincidencia inversa (esto es, que se rechace un paquete si la cadena coincide). De esta forma se pueden implementar numerosos filtros comunes, como filtrado por destino/origen (incluyendo puertos TCP/UDP) o incluso protocolo de la capa de aplicación.

La principal ventaja de este diseño es su simplicidad y su buen rendimiento, como se verá en la sección de validación experimental. Ahora bien, es un sistema poco flexible y los filtros pueden no ser correctos cuando las cabeceras de los paquetes varíen de tamaño: por ejemplo, no se podría hacer un filtro capaz de aceptar paquetes TCP con y sin cabecera VLAN (la cabecera VLAN añade bytes extra antes de la cabecera TCP).

Una posible solución a esa falta de flexibilidad sería permitir la creación de filtros anidados, de tal forma que se pueda implementar cualquier combinación lógica deseada. Sin embargo, eso implicaría una complejidad mayor en el bucle de recepción y por lo tanto un peor rendimiento. Además, sería más difícil traducir los filtros deseados a la combinación de cadenas y configuración necesarias para llevarlos a cabo.

La otra solución sería integrar el *driver* con los filtros BPF [10] integrados en el *kernel* Linux. Estos filtros se programan usando una sintaxis específica (por ejemplo, `host 192.168.1.1`) que se traduce a un lenguaje máquina específico. El código resultante se ejecuta en un intérprete optimizado dentro del *kernel*. Si bien es un sistema complejo para filtros sencillos, es apropiado para casos más genéricos al tener un lenguaje máquina fácilmente interpretable y además el compilador ya está preparado para los distintos protocolos y cabeceras de tamaño variable que puedan encontrarse.

Configuración de filtros "en vivo" Para evitar paradas innecesarias en la captura, el diseño del sistema de filtros permite la configuración "en vivo", esto es, sin dejar en ningún momento de capturar paquetes. Tal y como se comentaba en secciones anteriores, las necesidades de rendimiento del bucle de recepción impiden técnicas de sincronización especialmente complejas.

La activación de los filtros se controlará así con una variable atómica. Para activar los filtros, primero se rellenarán las estructuras correspondientes a través de una llamada `ioctl`, y después se pondrá a 1 la variable de control, de tal forma que los hilos sólo entren en el filtro cuando ya está listo para ser ejecutado.

Si el filtro ya se encuentra activado, primero se desactiva la variable de control y se esperan 1000 nanosegundos, para dar tiempo suficiente a todos los hilos a salir de la rutina de comprobación de filtros. Una vez pasado ese tiempo, se modifican los filtros y se reactivará la variable de control tal y como se ha descrito anteriormente.

3.3.2 Almacenamiento selectivo

En combinación con el filtrado, se pueden seleccionar ciertos datos del paquete a guardar para reducir la tasa efectiva a procesar posteriormente. Un primer enfoque sería implementar una limitación del tamaño de paquete capturado, tal y como se hace en otras herramientas de captura. Así, sólo se guardarían como mucho los n primeros bytes de cada paquete.

En la figura 3.4 se puede ver el efecto que tendrían las limitaciones de tamaño en la tasa efectiva de recepción que tendrían que soportar los sistemas de almacenamiento o aplicaciones de usuario que reciban el tráfico. Por ejemplo, para una limitación de tamaño de 100 bytes, con la que se podría guardar gran parte de la carga útil de cada paquete, la tasa efectiva disminuye rápidamente y en entornos reales (con un tamaño medio de paquete alrededor de los 500-600 bytes) se podría estar guardando en disco a una tasa por debajo de los 10 Gbps.

Una opción alternativa que permitiría guardar sólo la información más útil es un guardado únicamente de ciertos rangos de bytes de cada paquete. Así, podría extraerse la información necesaria para el programa que esté recibiendo tráfico y se aprovecharía al máximo la capacidad de la memoria y de los procesadores.

3.4 Herramientas adicionales y entorno de pruebas

Para poder probar y medir el rendimiento de la implementación, se desarrollan algunas herramientas y módulos adicionales. En primer lugar, se crea

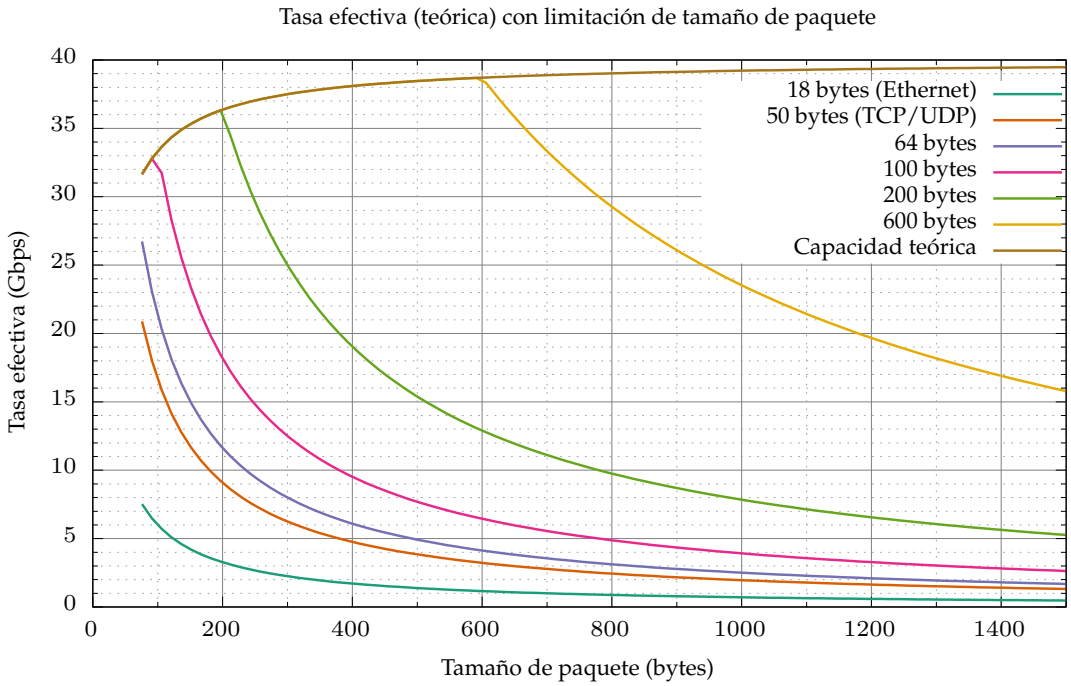

Figura 3.4: Tasa efectiva de recepción según el tamaño de paquete para distintos límites de tamaño de captura, incluyendo casos para capturar todas las cabeceras Ethernet o TCP/UDP.

un *script* encargado de compilar e insertar el *driver* en el sistema, comprobando que todos los parámetros y configuraciones se han cargado correctamente y que responde sin problemas. Adicionalmente, se pueden realizar pruebas de funcionamiento que comprueban que HPCAP no corrompe datos y que no se producen fallos en ciertas situaciones límite (por ejemplo, cuando los paquetes se ajustan exactamente al tamaño del *buffer*, que puede provocar problemas con el *padding*; o cuando se utilizan etiquetas VLAN que la tarjeta no reconoce correctamente).

Las estadísticas de tasa de recepción y pérdidas se obtienen extrayendo los datos internos de la tarjeta, y añadiendo los provenientes del *driver* (principalmente el número de paquetes descartados por espacio insuficiente en el *buffer* de copia). Esto permite controlar y medir el rendimiento de sistema de captura.

Por último, para facilitar el desarrollo, se incluye dentro del *driver* un módulo de medición de rendimiento (*profiling*) que mantiene estadísticas internas de funcionamiento, incluyendo los paquetes leídos del anillo de la tarjeta por cada llamada a la función de recepción, el número de veces que los hilos de recepción se encuentran con su segmento vacío, y la cantidad de descriptores devueltos en cada lote a la tarjeta.

Este módulo de medición es especialmente relevante porque permite la identificación de los puntos débiles del diseño y los cuellos de botella sin influir notablemente en el rendimiento ni modificar el comportamiento del *driver*. De hecho, fue muy útil para identificar problemas en la tarjeta de red y poder cambiar la configuración para optimizar la velocidad de recepción.

3.5 Conclusiones del capítulo

A lo largo de este capítulo se ha expuesto la arquitectura necesaria para recibir tráfico a 40 Gbps, usando múltiples hilos para mejorar el rendimiento. Conociendo el funcionamiento del *driver* y las condiciones en las que se ejecuta, se pueden implementar sistemas de sincronización simples y que afectan poco al rendimiento usando variables atómicas en lugar de otros métodos (*spinlock*, *mutex* o semáforos) que serían más lentos e impedirían alcanzar tasas altas de tráfico.

Además, se han descrito los pasos necesarios para integrar esa arquitectura con los *drivers* existentes, y se han planteado soluciones para poder reducir la cantidad de datos que se reciben en las aplicaciones cliente para reducir los requisitos de almacenamiento necesarios.

En el siguiente capítulo se pasará a comprobar el funcionamiento y la efectividad real del sistema diseñado.

4 VALIDACIÓN EXPERIMENTAL

En este capítulo se comprueba el rendimiento de la arquitectura propuesta, validando que supone una mejora frente al enfoque básico con un único hilo; y además se muestra la efectividad de las características adicionales para la reducción de los datos recibidos por los clientes.

En primer lugar, se detallará el entorno usado para desarrollar las pruebas. Después se describirán las pruebas realizadas de captura de tráfico sobre la tarjeta Mellanox, comparando los resultados según el número de hilos usados, y por último se presentarán los resultados de filtrado y almacenamiento selectivo de paquetes. En una sección adicional se mostrarán los resultados obtenidos con la tarjeta de Intel, que no son tan detallados al haber tenido problemas con el generador de tráfico.

4.1 Entorno experimental

	skadi	**einsrepo**
Procesador	2 × Intel Xeon E5-2620 2,40 GHz, 6 núcleos	Intel Xeon E5-1620 3,7 GHz, 4 núcleos
RAM	64 GB	32 GB
NIC	Mellanox MT27500 ConnectX-3	Intel XL710

Tabla 4.1: Especificaciones técnicas de los sistemas de prueba.

El entorno de pruebas consiste de dos servidores del grupo HPCN, *skadi* y *einsrepo*, cuyas especificaciones se detallan en la tabla 4.1. Estos serán los encargados de recibir el tráfico usando el driver de HPCAP.

Para maximizar el rendimiento, se configura apropiadamente la asignación de núcleos de CPU teniendo en cuenta la arquitectura NUMA (figura 4.1) de los servidores y los nodos a los que están conectados la tarjeta de red y el sistema de almacenamiento RAID. La distribución óptima [13] consiste en ejecutar los hilos de recepción en el mismo nodo NUMA al que se encuentra conectada la tarjeta de red y el hilo de volcado a disco en el nodo donde está conectado el RAID.

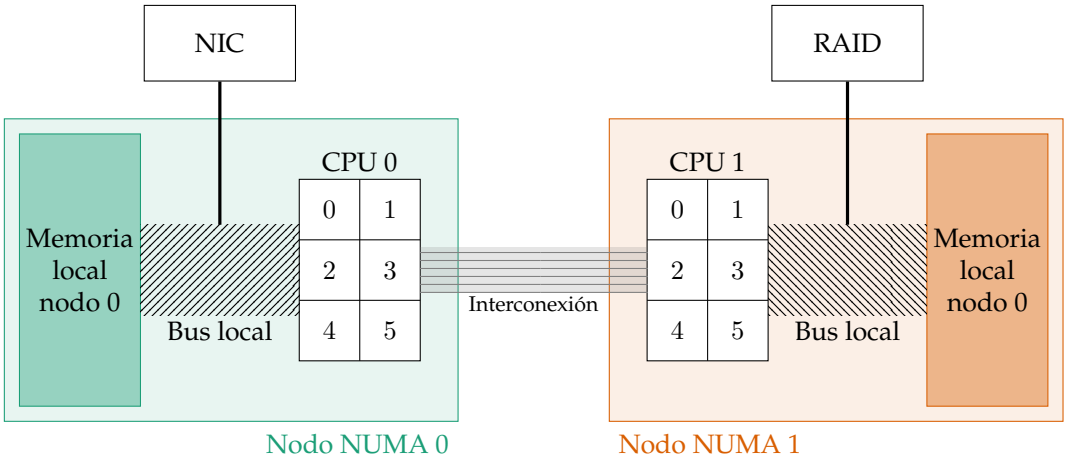

Figura 4.1: Arquitectura NUMA de los sistemas de pruebas.

Esta configuración se ajusta automáticamente con las herramientas de instalación desarrolladas para el *driver*, que detectan la distribución del equipo y ajustan la afinidad de los hilos de recepción de manera automática.

Para las pruebas ambos equipos se conectan directamente entre sí por una conexión de 40 Gbps: uno de ellos ejecutará un generador de tráfico y en el otro estará instalado HPCAP para la recepción de los datos. El generador de tráfico usado es *pktgen-DPDK*, basado en DPDK. Aunque para tamaños de paquete bajos no es capaz de saturar el enlace, es la mejor opción disponible para realizar las pruebas.

El principal problema de este entorno experimental es que impide la comparación de rendimiento de las dos tarjetas de Intel y Mellanox, ya que las especificaciones de los sistemas son distintas y por lo tanto no se puede distinguir entre los factores intrínsecos a la implementación y a la tarjeta y los que dependen del procesador usado. Aun así, cumple el propósito de servir como plataforma de pruebas y validación del rendimiento del *driver* desarrollado y de la arquitectura planteada.

4.2 Recepción de tráfico

4.2.1 Arquitectura básica: capacidades y limitaciones

Dadas las limitaciones del hardware, está claro que un único hilo de recepción no será suficiente para hacer la recepción a 40 Gbps. Aun así, es necesario realizar las pruebas en esta arquitectura simple para establecer una base sobre la que comparar y medir mejoras.

La prueba se realiza en el entorno descrito anteriormente. La tasa de envío del generador de paquetes se incrementa gradualmente hasta llegar al máximo que HPCAP es capaz de recibir sin perder ningún paquete. Para ello, se comprueban los contadores de pérdidas de la tarjeta, visibles desde la herramienta estándar `ethtool`.

Tal y como se ve en la figura 4.2, sólo se llega a la tasa máxima de captura a partir de los 1250 bytes de tamaño de paquete. Para tamaños pequeños se observa

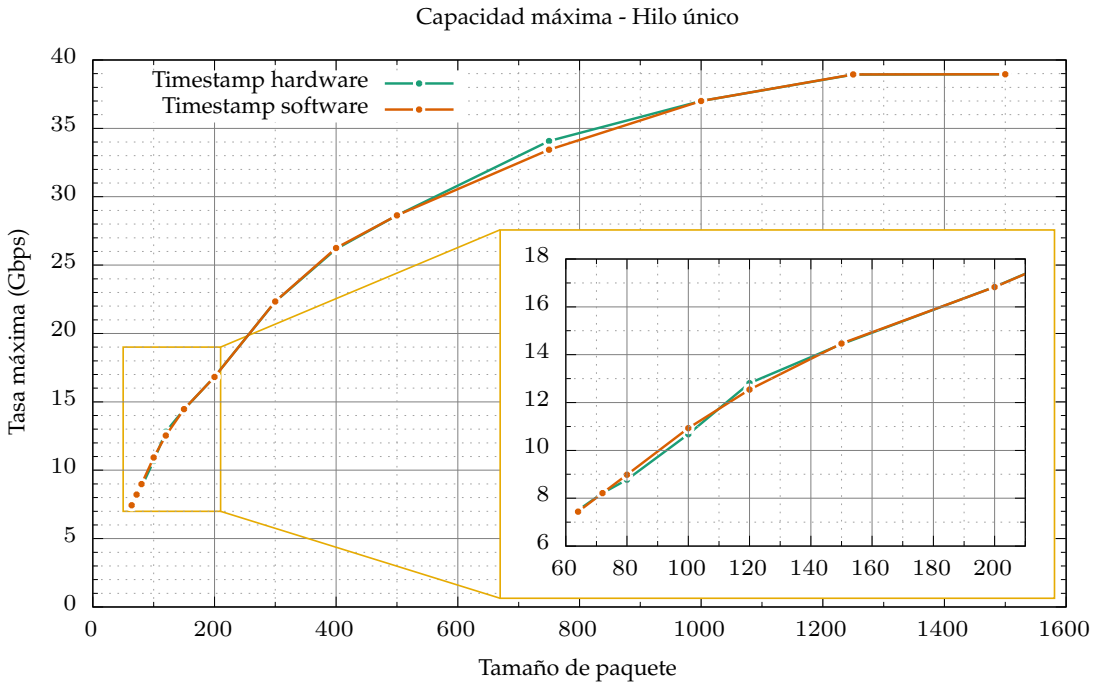

Figura 4.2: Una medida de la capacidad base de la tarjeta: tasa máxima que se alcanza sin perder tráfico en función del tamaño de paquete.

el mismo comportamiento que con la versión original de HPCAP [11], donde para tamaños cercanos al mínimo (64 bytes) no se llegaba a la tasa máxima de 10 Gbps. En nuestro caso, para ese mismo tamaño se llega a una tasa de 7.6 Gbps.

Así mismo, sobre esta arquitectura básica se ha comprobado el efecto de las marcas de tiempo (*timestamp*) obtenidas a través de la tarjeta (sólo disponible en la tarjeta de Mellanox) o a través del reloj software del *kernel* Linux. El rendimiento en ambos casos es prácticamente el mismo, lo que permite relajar los requisitos de ordenación de paquetes al poder confiar en la tarjeta para realizar el marcado.

En cualquiera de los dos casos, se observa que un hilo no es suficiente para conseguir una tasa aceptable de tráfico, por lo que es necesario diseñar un sistema de recepción distinto que pasamos a probar a continuación.

4.2.2 Arquitectura con múltiples hilos

A lo largo de esta sección se valida la arquitectura diseñada en la sección 3.2, comprobando que efectivamente es capaz de recibir a una tasa de tráfico suficiente y minimizando el desorden de paquetes.

El primer resultado es el del tráfico máximo que el sistema es capaz de recibir sin perder ningún paquete. El entorno de pruebas es el mismo que el descrito en la sección anterior.

Como se puede ver en la figura 4.3, el sistema alcanza la tasa teórica de 40 GbE a partir de los 600 bytes de tamaño, un resultado aceptable ya que el tamaño medio de paquete se encuentra en el entorno de los 600 bytes [7]. Para tamaños más bajos la capacidad del sistema baja considerablemente, llegando a los 12 Gbps para paquetes de tamaño mínimo (64 bytes).

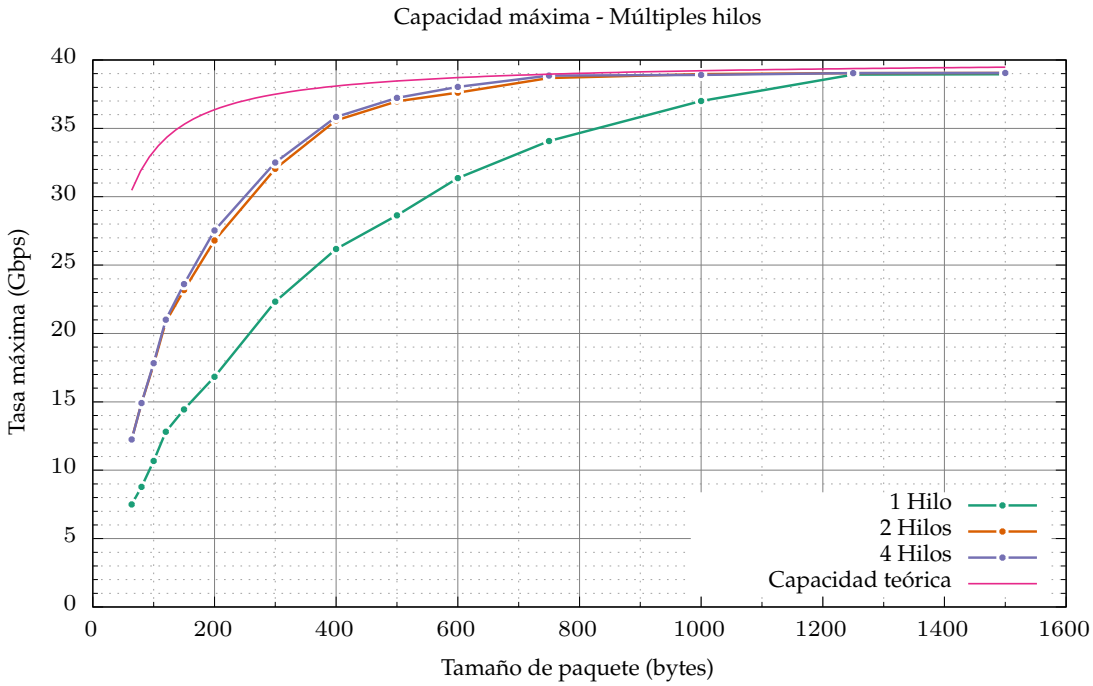

Figura 4.3: Tráfico máximo soportado por el diseño de recepción a 40Gbps para 1, 2 y 4 hilos. El gráfico incluye la capacidad máxima teórica del enlace Ethernet.

Los resultados según el número de hilos nos permiten verificar que el diseño funciona y que es capaz de mejorar la tasa de recepción con un único hilo. Además, dada la variación mínima de rendimiento entre el uso de dos y con cuatro hilos de recepción, se observa que la tarjeta Mellanox no es capaz de dar el total de la capacidad del enlace con un único anillo de recepción. Esta hipótesis está soportada además por el hecho de que, durante el desarrollo, se vio que el rendimiento dependía de manera crítica de la configuración de la tarjeta. De hecho, no se consiguió una mejora de velocidad con múltiples hilos hasta que no se configuró apropiadamente la tarjeta de red (ver sección 3.1.1 para los ajustes concretos).

Además de la tasa de recepción, se comprueba que la arquitectura no introduce un desorden excesivo. Cierto desorden en la captura es aceptable, ya que este se puede corregir usando las marcas de tiempo precisas que pone la tarjeta a cada paquete de llegada, aunque para facilitar los análisis es preferible que ese desorden no sea demasiado alto.

Para realizar las pruebas, se configuraron dos servidores de la misma forma que en secciones anteriores y se configuró *pktgen-DPDK* para marcar su orden de salida. En el servidor de captura se usó el driver HPCAP configurado para capturar únicamente los campos correspondientes de cada paquete, de tal forma que se pueda guardar en disco la información necesaria sin pérdidas.

En primer lugar se comprobó que con un único hilo se producía el comportamiento esperado, sin ningún tipo de desorden introducido. Esto permite confirmar la corrección de la implementación y asegurar que la tarjeta de red no introduce ningún desorden por sí misma.

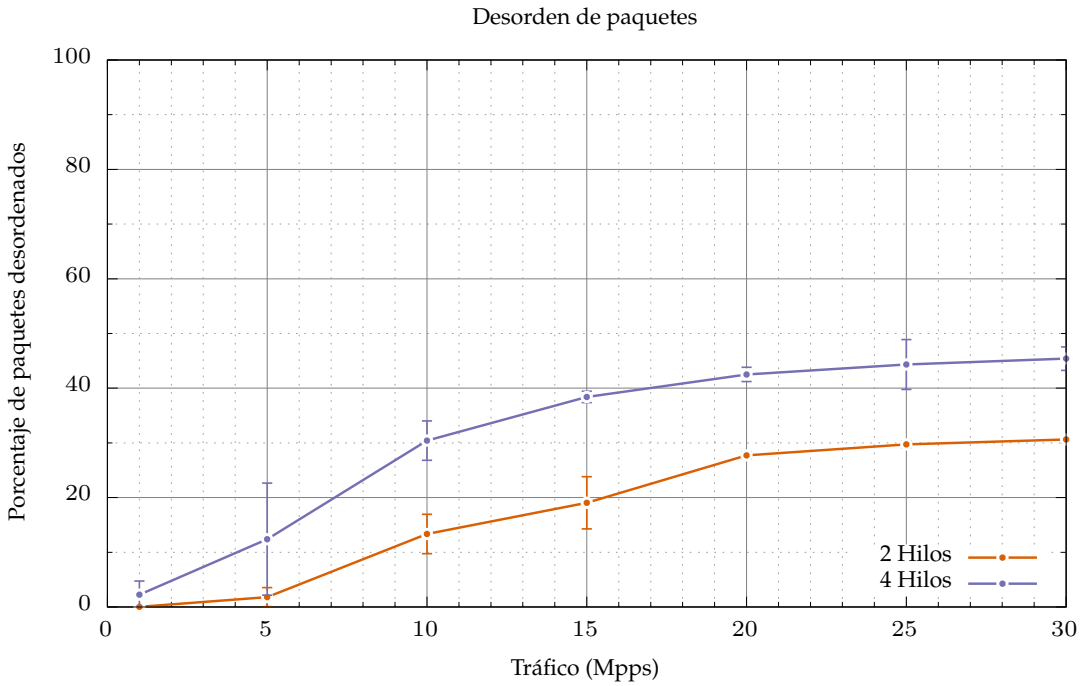

Figura 4.4: Porcentaje sobre el total de paquetes recibidos que se desordenan al pasar por el sistema de captura. Las barras de error muestran la variación de resultados al cambiar la velocidad y tamaño de paquete manteniendo el mismo *ratio* de paquetes por segundo.

Los resultados de la prueba completa se pueden ver en la figura 4.4, que muestra el porcentaje de paquetes desordenados según el tráfico, medido en millones de paquetes por segundo. El desorden se mantiene en valores aceptables para tasas de tráfico bajas y llega hasta el 30 % con tasas altas; tasas que por otra parte sólo se alcanzan con paquetes pequeños, una situación poco frecuente en entornos de captura reales.

También se puede observar que, tal y como se esperaba dado el diseño del sistema, el porcentaje de desorden aumenta al aumentar el número de hilos.

4.3 Captura y guardado de tráfico

Una vez comprobadas las capacidades del diseño para recibir datos de la tarjeta, la siguiente prueba medirá el rendimiento del sistema de escritura a memoria y la aplicación de guardado a disco.

El entorno de pruebas es el mismo que el planteado en pruebas anteriores, y se compara el rendimiento del *driver* al descartar los paquetes y al guardarlos en memoria RAM, en un sistema de archivos *tmpfs*. De esta forma se mide el rendimiento del sistema de sincronización de hilos para guardar los datos en memoria y se aíslan los efectos de velocidad de disco, cuya medida queda fuera del alcance de este trabajo.

Como era esperable, el rendimiento disminuye al guardar datos en el sistema de archivos en RAM, y de hecho no se llega a la tasa completa de 40 Gbps. Cabe destacar que la tasa obtenida es muy dependiente del tamaño del *buffer* de HPCAP. En estas pruebas se usó un *buffer* de 2 GB, que no se pudo ampliar

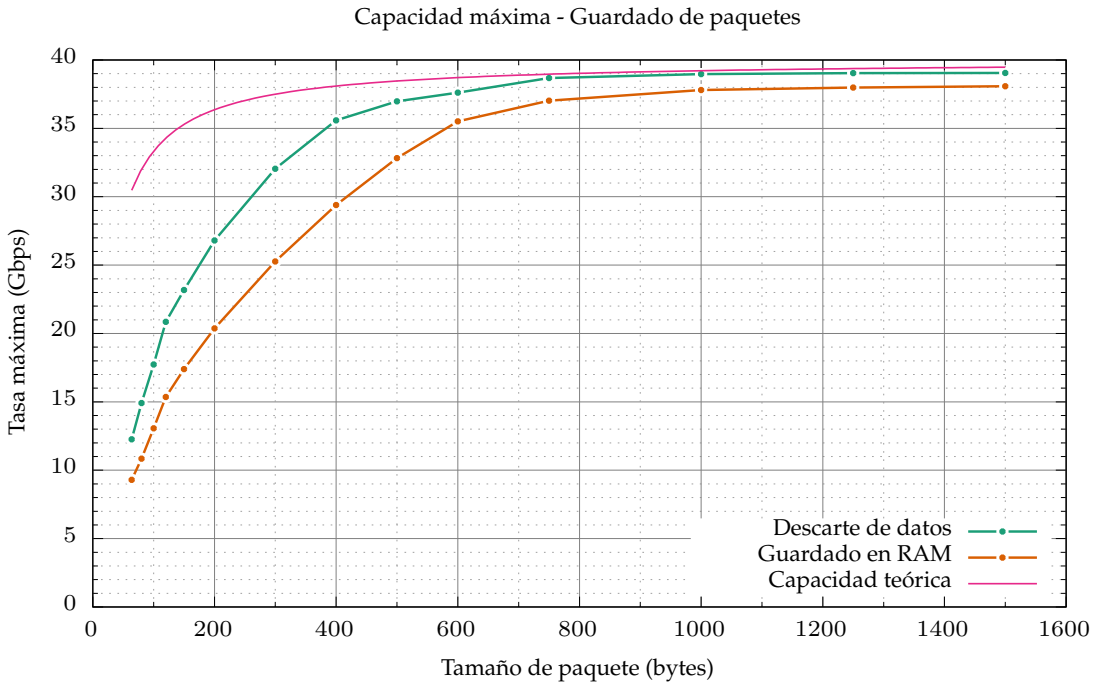

Figura 4.5: Comparación entre rendimiento descartando paquetes y guardándolos en un sistema de archivos en RAM.

debido a un *bug* en el *kernel* de Linux, concretamente en las rutinas encargadas de asignar direcciones virtuales a páginas de tipo *huge*. Durante el desarrollo, se creó y envió un parche[1] a Linux para corregir este problema, que ya ha sido aceptado.

4.3.1 Filtrado

Manteniendo el mismo entorno que en las pruebas de recepción, se configura un número de filtros determinado por cada ejecución con un tamaño fijo. Después se genera tráfico a diferentes tasas y tamaños de paquete, comprobando que la tasa de pérdidas no varíe con respecto al sistema sin filtros activados.

En esas pruebas no se han encontrado diferencias significativas de rendimiento: la variación de paquetes perdidos y tasa máxima de recepción es mínima y prácticamente inapreciable teniendo en cuenta la propia variabilidad de ambas medidas.

4.3.2 Almacenamiento selectivo

Para comprobar los efectos del almacenamiento selectivo, se realizan tres pruebas de recepción limitando el tamaño de paquete a 18, 64 y 200 bytes. El primer caso captura toda la cabecera a nivel Ethernet, el segundo caso captura cabeceras Ethernet, IP y de la capa de transporte (TCP/UDP). Por último, una captura de 200 bytes permitiría almacenar prácticamente cualquier cabecera de la capa de aplicación y una parte de la carga útil del paquete.

En la figura 4.6 se puede observar que el rendimiento mejora en todos los casos, y se alcanza la tasa máxima soportada por el *driver* desarrollado. Los mejores resultados se obtienen con los límites de 18 y 64 bytes, especialmente

[1] http://marc.info/?l=linux-mm&m=146114962421719&w=2

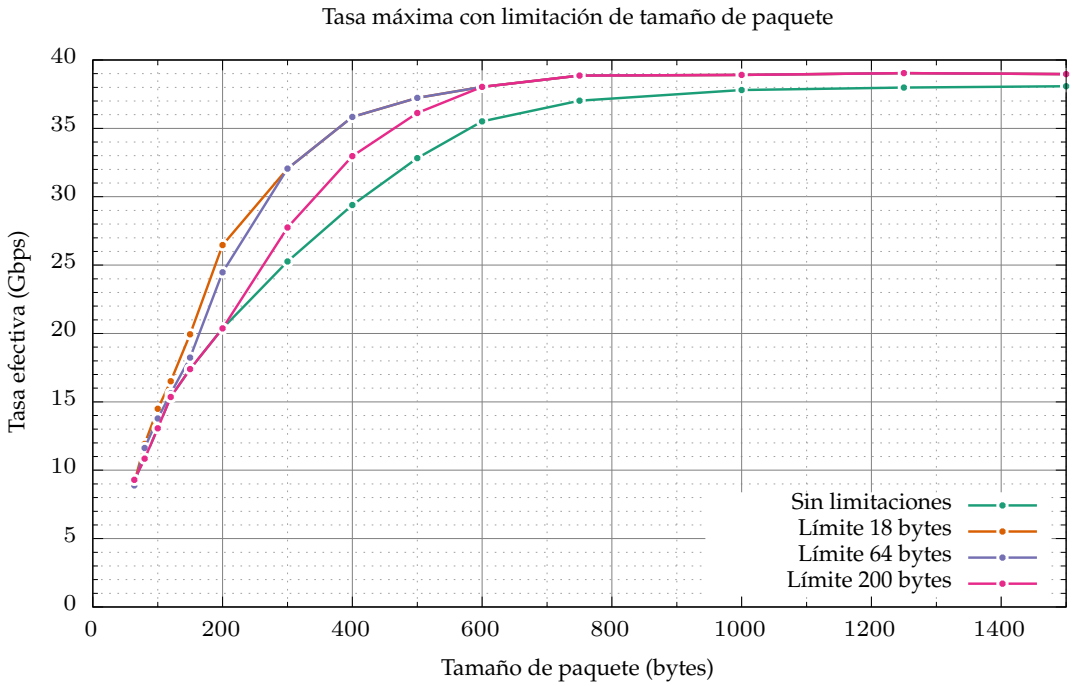

Figura 4.6: Tasa máxima de recepción guardando los datos en RAM y limitando el tamaño de paquete.

para tamaños de paquete pequeños. A partir de los 600 bytes por paquete, en los tres casos la tasa alcanzada es la misma: la máxima que era capaz de dar el sistema descartando todos los datos.

4.4 Recepción y almacenamiento de tráfico - Tarjeta Intel

Para la validación de la implementación sobre la tarjeta Intel se ha planteado el mismo entorno de pruebas que en secciones anteriores cambiando los roles: *skadi* se usa como generador de tráfico con la tarjeta Mellanox, y el receptor es *einsrepo* con la Intel XL710. Sin embargo, un problema con las tarjetas Mellanox y con el sistema de generación de tráfico *pktgen-DPDK* impide hacer las pruebas correctamente.

El generador no es capaz de controlar la tasa de generación de tráfico por falta de soporte de la tarjeta de Mellanox, por lo que las pruebas realizadas anteriormente no se pueden realizar. La alternativa en este punto sería pasar a generar a tasa máxima y medir el porcentaje de pérdidas, pero tampoco es una alternativa factible ya que para tamaños de paquete pequeños *pktgen-DPDK* no es capaz de saturar el enlace, de tal forma que las medidas de pérdidas no serían comparables ni útiles.

Las pruebas que se realicen para esta tarjeta son menos precisas que las realizadas para la Mellanox, y sólo se obtienen resultados significativos en una de ellas: la prueba de guardado en RAM de los datos. El resto de pruebas eran o bien imposibles de realizar por no poder controlar la tasa (desorden de paquetes)

o bien porque el generador no era capaz de dar una tasa suficientemente alta como para provocar pérdidas y medir la capacidad máxima del *driver*.

En el caso de la prueba de guardado en RAM de los datos sí se han podido provocar pérdidas en el *driver* que den idea de la capacidad máxima del sistema. Sin embargo, son medidas menos precisas que en la tarjeta Mellanox al cambiar la metodología: en lugar de aumentar progresivamente la tasa hasta provocar pérdidas, se genera tráfico y se mide la tasa a la que el *driver* está recibiendo paquetes. En general, esta tasa es menor que la capacidad máxima de recepción, ya que al introducir pérdidas baja el rendimiento de los hilos de recepción y de las rutinas del hardware.

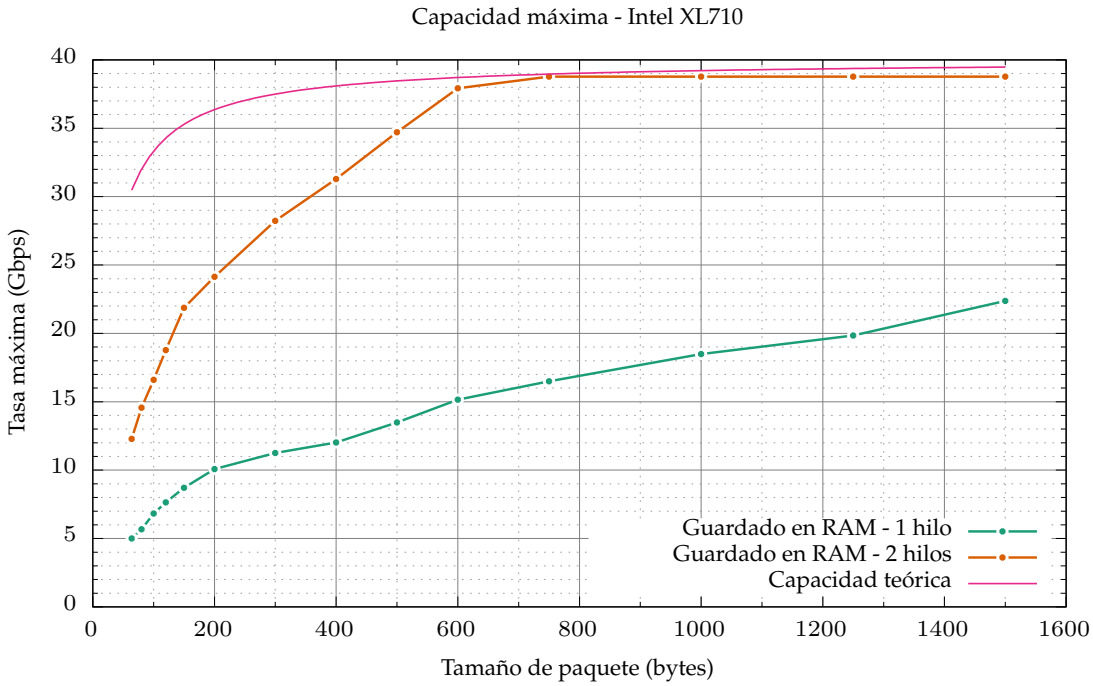

Figura 4.7: Tasa de recepción de tráfico con guardado en RAM de los datos para la tarjeta Intel XL710.

En cualquier caso, en la figura 4.7 se observa que la tasa obtenida es bastante alta para el guardado en RAM de los datos con dos hilos de recepción, llegando prácticamente a la tasa máxima con 600 bytes por paquete. Además, de nuevo se verifica la validez del diseño multihilo: la mejora de rendimiento al pasar de uno a dos hilos de recepción es muy significativa.

Los resultados son ligeramente mejores de los obtenidos para la tarjeta Mellanox (ver figura 4.5), aunque como se comentaba en la sección 4.1 los resultados no son comparables debido a las diferencias del hardware de los sistemas de prueba.

5 Conclusiones

En este Trabajo de Fin de Grado se ha conseguido desarrollar un sistema capaz de recibir tráfico a 40 Gbps para paquetes de tamaño medio (600 bytes) usando un diseño multihilo capaz de aprovechar las arquitecturas multiprocesador de sistemas estándar de alta gama, sin necesidad de usar hardware especializado y por lo tanto manteniendo un coste bajo para la monitorización. Además, se han desarrollado soluciones para reducir la cantidad de datos a recibir por aplicaciones cliente aumentando así la tasa máxima que las aplicaciones cliente serán capaces de analizar o de almacenar en disco. Todo ello se ha realizado de manera portable, manteniendo una misma base de código para tarjetas de diferentes fabricantes (Mellanox y Linux).

Para ello, en primer lugar los puntos débiles de soluciones actuales de captura de tráfico para desarrollar un sistema capaz de recibir a 40 Gbps. Se ha explorado la arquitectura del sistema de red de Linux, viendo que es insuficiente para sistemas de alto rendimiento por el uso de interrupciones y de copias múltiples e innecesarias. Por otra parte, una revisión del estado del arte en sistemas de captura indica que no hay apenas herramientas de captura a 40 Gbps, y que además pocas se enfocan en un marcado de tiempo preciso y en el mantenimiento del orden de recepción de los paquetes.

Se utiliza de base el trabajo ya desarrollado de Víctor Moreno en su tesis doctoral, HPCAP, capaz de recibir a 10 Gbps en tarjetas de red de Intel. Este motor de captura desactiva las interrupciones y realiza un *polling* continuo al anillo de recepción de la tarjeta, un enfoque mucho más eficiente para tasas altas de tráfico. Además, cuenta con una API para aplicaciones cliente que quieran recibir los paquetes, y está preparado para almacenar el tráfico directamente y de forma eficiente a disco usando técnicas de *padding*.

Para que HPCAP sea capaz de recibir a 40 Gbps es necesario un cambio de arquitectura: un único hilo no es capaz de recibir a esa tasa con los procesadores actuales. Esa arquitectura se desarrolla en el capítulo 3, usando técnicas de concurrencia simples y de alto rendimiento, principalmente basadas en variables atómicas, y el conocimiento sobre el entorno de funcionamiento de HPCAP que

permite evitar otras técnicas más lentas y que impedirían la recepción a alto rendimiento.

Así pues, la arquitectura es capaz de lanzar varios hilos que leen al mismo tiempo de un anillo de recepción usando una división y asignación de segmentos fijos por cada hilo, y devolviendo los descriptores a la tarjeta para su reutilización de forma ordenada. Además, se usan variables atómicas y un algoritmo de *padding* para que todos los hilos escriban los paquetes en un único *buffer* y se pueda trabajar con él a nivel de bloque, mucho más eficiente si el objetivo es guardar los datos a disco.

Por otra parte y dado que a 40 Gbps se puede generar una cantidad de datos demasiado grande para sistemas de almacenamiento comunes, se desarrollan dos métodos capaces de reducir los datos a tratar con pérdida de información mínima: por un lado, un algoritmo de filtrado que reduzca al máximo el tiempo de comprobación de cada filtro; y por otro funciones de almacenamiento selectivo que permitan copiar y guardar sólo la información necesaria de cada paquete.

Además, el desarrollo se hace de manera portable para tarjetas Intel y Mellanox: el núcleo del código HPCAP es común a los dos *drivers* de cada fabricante, y sólo se realizan algunas modificaciones en cada caso específico, modificaciones que dependen del hardware y del funcionamiento de cada tarjeta.

La validación experimental demuestra que la arquitectura es efectiva, pudiendo recibir tráfico a 40 Gbps para paquetes suficientemente grandes (alrededor de 600 bytes) en ambas arquitecturas. Además, no se introduce un desorden excesivo en la captura, que era uno de los objetivos iniciales del trabajo.

Las características adicionales para reducción de los datos a guardar también funcionan apropiadamente: el filtrado de paquetes no afecta de manera apreciable al rendimiento de la captura, y el almacenamiento selectivo permite aumentar la tasa a la que es capaz de recibir el *driver*.

El entorno de pruebas creado también ha resultado suficiente para las pruebas realizadas, aunque no haya sido capaz de generar tráfico suficiente como para saturar el enlace 40 GbE. Las herramientas adicionales han sido útiles para preparar ese entorno y configurar el sistema aprovechando al máximo la arquitectura multiprocesador.

Por último, el desarrollo muestra que mantener un núcleo común y modificaciones puntuales en los *drivers* base es un enfoque factible y efectivo para el soporte de tarjetas de diferentes fabricantes, y facilita el desarrollo y soporte del sistema.

Como trabajo futuro, se podrían explorar más caminos para mejorar el filtrado y la selección de datos. Por ejemplo, se podría implementar filtrado de paquetes ASCII [16], una técnica rápida para capturar sólo paquetes con protocolos de texto plano, que podría aumentar la efectividad de la captura descartando todos los paquetes innecesarios con alta precisión.

También se podrían implementar limitaciones de tamaño dinámicas que varíen según la tasa de recepción, para maximizar los recursos en los que el tráfico no sea constante. Siguiendo con el filtrado, se podría comprobar si la

implementación de BPF para permitir filtros más complejos es práctica y no disminuye en exceso el rendimiento del *driver*.

En último lugar, sería necesario mejorar el entorno de pruebas con un generador de tráfico dedicado a 40 Gbps, capaz de saturar el enlace para cualquier tamaño de paquete y de generar flujos, de tal forma que se pueda estudiar el desorden de paquetes dentro del mismo flujo.

BIBLIOGRAFÍA

[1] *NAPI*, 2009. http://www.linuxfoundation.org/collaborate/workgroups/networking/napi.

[2] *DPDK: Programmer's Guide*, April 2016. http://dpdk.org/doc/pdf-guides/prog_guide-16.04.pdf.

[3] U. Alonso Camaró and J. Baudy. Linux packet_mmap. Technical report, Linux Foundation, 2016.

[4] N. Division. *Intel Ethernet Controller XL710 Datasheet*. Intel, March 2016. http://www.intel.es/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf.

[5] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 275–287. ACM, 2015.

[6] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2016. http://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[7] W. John and S. Tafvelin. Analysis of internet backbone traffic and header anomalies observed. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 111–116. ACM, 2007.

[8] A. Krizhanovsky. Lock-free multi-producer multi-consumer queue on ring buffer. *Linux Journal*, 2013(228):4, 2013.

[9] B. H. Leitao. Tuning 10Gb network cards on Linux. In *Proceedings of the 2009 Linux Symposium*. Citeseer, 2009.

[10] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2. USENIX Association, 1993.

[11] V. Moreno. Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks. Master's thesis, Escuela Politécnica Superior UAM, 2012.

[12] 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.

[13] 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):221–234, 2014.

[14] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[15] L. Rizzo, L. Deri, and A. Cardigliano. 10 Gbit/s line rate packet processing using commodity hardware: Survey and new proposals, 2012.

[16] V. Uceda, J. Ramos, J. L. García-Dorado, J. Aracil, et al. Selective capping of packet payloads at multi-Gb/s rates. 2015.