UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

FACULTAD DE
INGENIERÍA

Programabilidad de Red aplicada al Monitoreo mediante In-band Network Telemetry

Informe de Proyecto de Grado presentado por

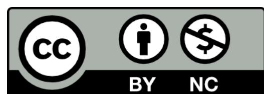
Joaquín Mezquita, Nicolas Temciuc, Joaquín Tomás

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Eduardo Grampín
Leonardo Alberro

Montevideo, 7 de mayo de 2026



Programabilidad de Red aplicada al Monitoreo mediante In-band Network Telemetry por Joaquín Mezquita, Nicolas Temciuc, Joaquín Tomás tiene licencia CC Atribución - No Comercial 4.0.

Agradecimientos

Este proyecto se encuentra ubicado en el tramo final de un camino que es producto de esfuerzos colectivos, no de conquistas individuales.

En ese sentido queremos agradecer en primer lugar a nuestras familias por el apoyo (material y espiritual), no solamente durante el desarrollo de este proyecto de grado, sino que durante el transcurso de la toda la carrera.

A los amigos, por el aguante.

A Eduardo Grampín y Leonardo Alberro, por su guía constante y por asegurarse de que este proyecto se mantuviera siempre en rumbo.

A Javier Baliosian, por ofrecernos una bajada a tierra en lo relativo a casos de uso concretos para esta investigación.

A todos ellos: muchas gracias.

Resumen

El presente proyecto aborda el diseño e implementación de una plataforma de monitoreo de red basada en *In-band Network Telemetry (INT)*, una técnica que permite recolectar información de desempeño directamente desde el plano de datos de la red. A diferencia de los mecanismos tradicionales de observación como *Simple Network Management Protocol (SNMP)* o *Remote Network Monitoring (RMON)*, INT inserta información de telemetría dentro de los propios paquetes de datos, proporcionando una visión precisa del comportamiento de la red.

Se implementa un INT *Sink* -el nodo encargado de recibir, procesar y reportar los metadatos generados por los dispositivos INT a lo largo del camino de los paquetes- sobre una *Smart Network Interface Card (SmartNIC) Netronome Agilio CX 2×10 GbE*, que combina un *pipeline* programado en *P4 (Programming Protocol-independent Packet Processors)* con rutinas en *Micro-C* ejecutadas directamente sobre los microprocesadores del dispositivo. Esta arquitectura híbrida permite interpretar los encabezados INT en el plano de datos, almacenar la información recolectada en estructuras en memoria y exportarla hacia el sistema anfitrión (*host*).

El proyecto extiende la plataforma presentada en un trabajo previo -que implementa el INT *Sink* directamente en una SmartNIC- en dos sentidos: (i) implementando nuevos mecanismos de instrumentación en la capa de aplicación, mediante la incorporación de campos que permiten identificar y correlacionar solicitudes y respuestas posibilitando la estimación posterior de tiempos de servicio o latencias de aplicación; y (ii) implementando un entorno de visualización y análisis construido sobre el *stack* ELK (*Elasticsearch*, *Logstash* y *Kibana*), lo que permite examinar las métricas recolectadas de forma flexible y dinámica en el *host*.

Además, se desarrolla una prueba de concepto de un nodo intermedio INT (*Transit Hop*) para evaluar la viabilidad de implementar el mismo con el *hardware* (*SmartNIC*) anteriormente mencionado.

El conjunto de estas implementaciones constituye un entorno experimental para el estudio de técnicas de telemetría embebida, combinando la programabilidad del plano de datos con herramientas modernas de análisis. La plataforma resultante aporta insumos para futuras investigaciones orientadas a la correlación entre métricas de red y métricas de aplicación, con potencial aplicación en entornos de *cloud computing*.

Palabras clave: Monitoreo de red, Programación del plano de datos, Dispositivos de red programables, *In-Band Network Telemtry*, Telemetría, Telemetría embebida, *Cloud computing*

Lista de acrónimos

ALU Arithmetic Logic Unit. 10

API Application Programming Interface. 8, 12, 13, 30

ASIC Application-Specific Integrated Circuit. 8, 9, 13, 14

BMv2 Behavioral Model. x, 13, 14, 54–56, 76

CLI Command Line Interface. 8

DPDK Data Plane Development Kit. 29, 32, 64

DSCP Differentiated Services Code Point. 21–23

DSL Domain Specific Language. 11

ELK Elasticsearch, Kibana y Logstash. ix, x, 2, 3, 29, 31–36, 39, 46, 48, 50, 52, 58, 72, 75, 76

FPGA Field-Programmable Gate Array. 13

ICMP Internet Control Message Protocol. 20

INT In-band network telemetry. ix, x, 1–3, 5, 14–27, 29, 33–45, 47, 49–59, 61–73, 75–78

MAT Match-Action Table. 10

MIB Management Information Base. 6

MTU Maximum transmission unit. 20, 24

NIC Network Interface Card. 13, 32

NMS Network Management Systems. 6

NPT Next Protocol Type. 21, 23

NTA Network Traffic Analysis. 5

P4 Programming protocol-independent packet processors. ix, 1–3, 5, 9, 11–14, 27, 30, 32–34, 36, 40–42, 44, 49, 54, 59, 63, 64, 67, 68

PISA Protocol-Independent Switching Architecture. ix, 5, 9–11

PSA Portable Switch Architecture. 14

PTP Precision Time Protocol. 14

RMON Remote network monitoring. 1, 6, 34

SDK Software Development Kit. 29, 30, 32, 44, 45, 54, 76

SDN Software-defined networking. 8, 9

SNMP Simple Network Monitoring Protocol. 1, 6, 34

TCP Transmission Control Protocol. ix, 21, 22, 35, 40, 41, 43, 49, 62

UDP User Datagram Protocol. ix, 21–23, 35, 40, 41, 43, 49, 62

Índice general

1. Introducción **1**

2. Marco teórico **5**
2.1. Monitoreo de redes . . . 5
2.1.1. Técnicas de monitoreo y análisis . . . 6
2.2. Dispositivos de red programables . . . 7
2.3. Programabilidad del plano de datos . . . 9
2.3.1. Protocol-Independent Switching Architecture (PISA) . . . 9
2.3.2. El lenguaje P4 . . . 11
2.4. In-band Network Telemetry (INT) . . . 14
2.4.1. Arquitectura general de INT . . . 15
2.4.2. Modos de operación . . . 16
2.4.3. ¿Qué se monitorea? . . . 17
2.4.4. *Headers* INT . . . 19
2.4.5. INT sobre TCP/UDP . . . 21
2.4.6. Formato de *INT-MD* Metadata Header . . . 23
2.5. Conclusiones . . . 27

3. Plataforma y tecnologías utilizadas **29**
3.1. *SmartNIC Netronome Agilio CX* . . . 29
3.2. ELK . . . 31
3.3. DPDK y *Moongen* . . . 32
3.4. Conclusiones . . . 32

4. Arquitectura e implementación **33**
4.1. Arquitectura general de la plataforma desarrollada . . . 34
4.1.1. Motivación y objetivos del diseño . . . 34
4.1.2. Descripción de la topología . . . 35
4.2. Diseño e implementación del INT *Sink* . . . 36
4.2.1. Procesamiento en P4 . . . 40
4.2.2. Procesamiento en Micro-C . . . 43
4.2.3. Comunicación con el host y exportación de datos . . . 46
4.2.4. Extensiones para casos de uso personalizados . . . 48
4.3. Sistema de visualización y almacenamiento . . . 50

4.3.1. Diseño del *pipeline* de monitoreo 50
4.3.2. ¿Por qué ELK? 52
4.4. Prueba de concepto de nodo intermedio INT 52
4.4.1. Funcionalidades de un nodo intermedio INT 53
4.4.2. Limitaciones en la *SmartNIC Netronome* 53
4.4.3. Implementación de la prueba de concepto en BMv2 . . . 54
4.5. Consideraciones y análisis de la arquitectura 56
4.5.1. Escalabilidad de la plataforma 57
4.5.2. Limitaciones del entorno 58
4.6. Conclusiones 59

5. Evaluación experimental 61
5.1. Entorno de evaluación 61
5.1.1. Generación de tráfico 61
5.2. *Throughput* 63
5.2.1. *Throughput* vs cantidad de nodos/instrucciones INT . . . 64
5.2.2. Posibles causas de degradación de *throughput* 66
5.2.3. Ensayos de optimización del *pipeline* 67
5.3. Latencia del procesamiento 68
5.4. Precisión de las métricas INT 69
5.5. Validación de la capa de visualización 72
5.6. Conclusiones 73

6. Conclusiones y Trabajo Futuro 75
6.1. Evaluación de los resultados alcanzados 75
6.2. Comparación entre objetivos planteados y resultados alcanzados 76
6.3. Contribuciones 76
6.4. Posibles casos de uso 77
6.5. Líneas de trabajo a futuro 77
6.6. Conclusión final 78

A. Bloques de código y pseudocódigo 83
A.1. Pseudocódigo *Micro-C Sink* 83
A.2. Estructuras principales en memoria 84
A.3. Pseudocódigo *Host_ Reader* 85

B. Guía de uso de la plataforma desarrollada 87

Capítulo 1

Introducción

Las redes de comunicación modernas generan volúmenes masivos y heterogéneos de tráfico que requieren mecanismos de observación y análisis cada vez más sofisticados. El monitoreo de red tiene como objetivo proporcionar retroalimentación sobre el estado de los dispositivos y de las conexiones entre ellos. Estas mediciones son útiles para detectar anomalías, anticipar fallos y prevenir comportamientos indeseados que puedan afectar el funcionamiento esperado del sistema.

Las metodologías tradicionales de monitoreo, como SNMP [21], *NetFlow* [6] o RMON [49], fueron efectivas en redes de menor escala, pero resultan insuficientes ante la complejidad, velocidad y dinamismo de las infraestructuras actuales [50]. Esas soluciones suelen presentar limitaciones en precisión, escalabilidad y latencia en la recolección de información, además de depender de mecanismos de sondeo que pueden consumir ancho de banda y no ofrecer una visión en tiempo real del estado interno de la red.

El reciente avance de los dispositivos de red programables [4] ha impulsado un nuevo enfoque de monitoreo denominado *In-band Network Telemetry (INT)* [36] [25]. La utilización de *switches* programables permite incorporar metadatos que contienen métricas de la red directamente en los paquetes generados por la capa de aplicación, es decir, en el tráfico de producción. A medida que estos paquetes atraviesan la red, cada dispositivo que soporta INT inserta sus metadatos, hasta que finalmente estos se extraen en un punto específico -el INT *Sink*- para ser reportados a una entidad de control encargada del monitoreo.

En este marco, el presente trabajo se enfoca en la implementación de un sistema de monitoreo basado en P4 [3] que utiliza INT para la recolección de métricas de red, complementado con mecanismos adicionales para incorporar metadatos personalizados que permitan medir con precisión los tiempos entre solicitudes y respuestas. Este despliegue experimental se realiza sobre *SmartNICs Netronome Agilio CX 2×10GbE*[1], en el entorno del proyecto de investigación *Smartlab*[2], y busca: (i) proveer una plataforma para que los operadores

[1] https://netronome.com/agilio-smartnics/

[2] El proyecto *Smartlab* es un laboratorio de red programable implementado con *Smart-*

de red puedan acceder fácilmente a información sobre el estado de la red y el tráfico en ella; y (ii) proveer un mecanismo que permita generar métricas que, en etapas futuras, puedan servir como insumo para modelos de *machine learning* orientados al análisis de comportamiento y a la gestión inteligente de recursos en entornos de *cloud computing* [43].

Este trabajo toma como referencia el artículo "*A SmartNIC-Accelerated Monitoring Platform for In-band Network Telemetry*" [19], donde se propone una plataforma de monitoreo basada en INT, en particular la implementación de un INT *Sink*. En dicha plataforma, el procesamiento de los paquetes INT se realiza utilizando una combinación de un *pipeline* P4, que se encarga de interpretar los encabezados, y algoritmos en *Micro-C* ejecutados en la *SmartNIC*, que realizan operaciones complejas como agregación, detección de eventos y notificación. Esta partición del procesamiento permite acelerar las operaciones complejas que serían difíciles de implementar únicamente en P4.

El objetivo general de este trabajo es reproducir la plataforma de monitoreo desarrollada en el artículo de referencia. Los objetivos específicos son los siguientes:

- Extender la plataforma mencionada incorporando metadatos adicionales orientados a correlacionar *requests* y *responses*, con el fin de habilitar análisis relacionados al tiempo de respuesta en escenarios basados en microservicios y entornos de *cloud computing* [43].

- Implementar un sistema de visualización utilizando el *stack* ELK [13], que permita explorar y analizar de forma integrada los datos exportados por la plataforma.

- Desarrollar una prueba de concepto de un nodo INT intermedio, con el fin de evaluar la factibilidad de su implementación en placas *Netronome Agilio CX 2×10GbE*.

Asimismo, con el fin de favorecer la reproducibilidad y transparencia del trabajo, todos los artefactos desarrollados -incluyendo los programas P4, el código *Micro-C*, los *scripts* de construcción de trazas, las configuraciones del *stack* ELK y el material utilizado para las pruebas- se encuentran en un repositorio público [41].

En términos generales, los resultados obtenidos muestran que la plataforma desarrollada reproduce con éxito la arquitectura propuesta en el trabajo de referencia, integrando un INT *Sink* funcional en una *SmartNIC Netronome Agilio CX* y extendiéndolo con metadatos adicionales. La incorporación de un *pipeline* de análisis y visualización basado en ELK permitió explorar los datos exportados y construir herramientas de observación sobre el tráfico instrumentado.

NICs de bajo costo, llevado a cabo por el grupo *MINA (Network Management / Artificial Intelligence)* del Instituto de Computación de la Facultad de Ingeniería, Universidad de la República.

La evaluación experimental evidencia que la plataforma puede procesar varios millones de paquetes por segundo y reportar métricas INT con precisión cuando no hay pérdida de paquetes en el ingreso al *pipeline*. También muestra que, bajo tasas elevadas, el rendimiento se ve degradado.

Finalmente, la prueba de concepto del nodo INT intermedio permitió determinar la no factibilidad de implementarlo en el *hardware* anteriormente descrito, a la vez que proporcionó una implementación en software que puede servir como referencia o guía para desarrollos futuros orientados a completar un dominio INT más amplio.

Este documento se organiza de la siguiente manera:

- En el Capítulo 2 se presenta el marco teórico, donde se describen los conceptos fundamentales relacionados con el monitoreo de redes, la programabilidad del plano de datos y la especificación de INT.
- En el Capítulo 3 se describen las tecnologías utilizadas, incluyendo la *SmartNIC* Netronome Agilio CX, su entorno de programación y el *stack* ELK.
- El Capítulo 4 desarrolla la arquitectura e implementación de la plataforma propuesta, detallando el diseño del INT *Sink*, el procesamiento en P4 y *Micro-C*, el sistema de visualización y la prueba de concepto del nodo INT intermedio, junto con un análisis de sus limitaciones y diferencias respecto al trabajo de referencia.
- El Capítulo 5 contiene la evaluación experimental, describiendo el entorno de pruebas, la metodología aplicada y los resultados obtenidos en términos de rendimiento y precisión de las métricas INT.
- Finalmente, el Capítulo 6 expone las conclusiones generales, se comparan los objetivos planteados con los resultados alcanzados, se detallan los aportes del trabajo y se discuten posibles líneas de investigación futura.

En este documento, las palabras en inglés o en "espanglish" se presentan en *cursiva*. Los acrónimos se despliegan al menos en su primera mención y se encuentran referenciados en la lista de acrónimos. Se utiliza el formato `monoespaciado` para referenciar elementos de código, como variables, funciones, comandos, módulos, etc.

Capítulo 2

Marco teórico

En este capítulo se reúnen los conceptos fundamentales necesarios para comprender el diseño y la implementación de la plataforma desarrollada, así como el contexto en el que esta se enmarca. La estructura del capítulo se organiza de la siguiente manera:

- En la sección 2.1 se abordan los conceptos básicos del monitoreo de redes, incluyendo un breve repaso de las técnicas clásicas.
- En la sección 2.2 se introducen los dispositivos de red programables y las capacidades de intervención que habilitan en los distintos planos de una arquitectura de red.
- En la sección 2.3 se profundiza en la programabilidad del plano de datos, introduciendo la arquitectura PISA (Protocol-Independent Switching Architecture) y el lenguaje P4 (Programming Protocol-Independent Packet Processors).
- En la sección 2.4 se describe en detalle el funcionamiento de INT, que define el mecanismo de recolección de métricas abordado en este trabajo.

2.1. Monitoreo de redes

El monitoreo de redes es el proceso de observar, recolectar y analizar información sobre el tráfico y el comportamiento de los dispositivos que componen una infraestructura de comunicación. Su objetivo es garantizar un funcionamiento seguro y eficiente, detectando fallos, congestiones o comportamientos anómalos antes de que afecten el rendimiento general [20] [18].

El análisis del tráfico (*Network Traffic Analysis*, NTA) permite obtener una visión detallada del estado interno de la red, evaluando métricas como la ocupación de colas, la latencia entre nodos o el ancho de banda disponible. Este tipo de mediciones constituye la base para la optimización, el diagnóstico y la gestión

inteligente de redes modernas, particularmente en entornos de gran escala como centros de datos, infraestructuras en la nube o redes definidas por software [50].

2.1.1. Técnicas de monitoreo y análisis

Las técnicas de monitoreo pueden clasificarse de manera general en dos categorías: (i) aquellas basadas en funcionalidades incorporadas en los enrutadores (*Router-Based Techniques*) y (ii) aquellas que dependen de componentes externos o software especializado (*Non-Router-Based Techniques*) [5]. Entre las primeras se encuentran mecanismos ampliamente utilizados como SNMP, RMON y *NetFlow*, mientras que entre las segundas se destacan los métodos de monitoreo activo y pasivo, que ofrecen una visión complementaria del estado de la red.

Uno de los protocolos clásicos en esta área es el *Simple Network Management Protocol* (SNMP) [17], empleado para supervisar y gestionar dispositivos de red. Su funcionamiento se articula en torno a agentes instalados en los dispositivos, sistemas gestores de red (NMS) y una Base de Información de Gestión (MIB) que describe de forma estructurada los parámetros consultables o modificables. Las operaciones básicas de SNMP, como *GET*, *SET* y *TRAP*, permiten obtener métricas, ajustar configuraciones o recibir notificaciones de eventos relevantes [23, 30]. En la Figura 2.1 se muestra un ejemplo de monitoreo basada en este protocolo.

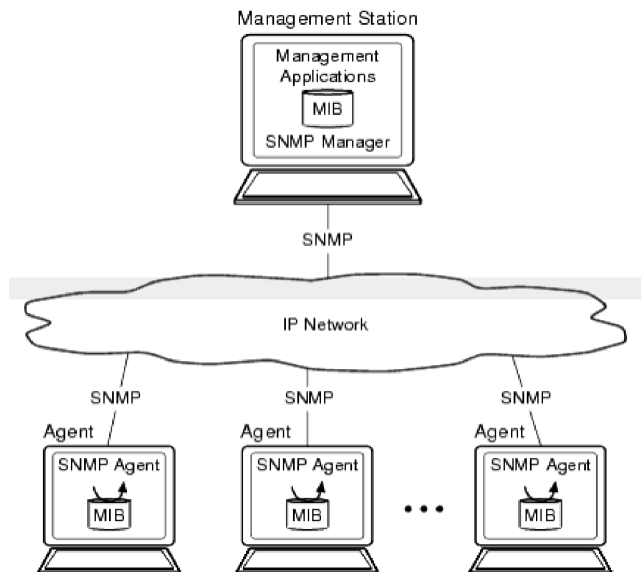


Figura 2.1: Red con SNMP [30].

Complementando a SNMP, el estándar RMON, definido en los RFC 1271

[47], 1757 [48] y 2819 [49], permite desplegar sondas que recopilan y analizan tráfico de forma autónoma. Este diseño descentralizado distribuye parte de la inteligencia del monitoreo hacia los nodos, evitando la sobrecarga del sistema gestor central y facilitando el diagnóstico remoto [16].

Otra herramienta ampliamente empleada es *NetFlow*, desarrollada por Cisco para caracterizar flujos de tráfico IP mediante la observación del comportamiento en las interfaces de red. Su arquitectura -compuesta por un módulo de recolección local (*Flow Caching*), un colector de flujos (*Flow Collector*) y un analizador (*Data Analyzer*)- permite obtener información como origen y destino del tráfico, volumen transferido, puertos, protocolo o marcas de tiempo [5], lo cual resulta clave en tareas de optimización, planificación y detección de congestión.

Además de estas técnicas basadas en los dispositivos de red, existen enfoques externos complementarios. El monitoreo activo introduce paquetes de prueba (*probes*) para evaluar métricas como retardo, pérdida o *jitter*, empleando herramientas como *ping*, *traceroute* o *iperf* [5]. Aunque este enfoque permite obtener mediciones precisas y controladas, introduce tráfico adicional que podría interferir con el tráfico real.

Por su parte, el monitoreo pasivo observa únicamente el tráfico real sin generar paquetes adicionales. Este enfoque facilita medir tasas de tráfico, composición por protocolos y tiempos entre llegadas mediante herramientas de captura como *packet sniffers* [5]. Si bien evita la sobrecarga propia del monitoreo activo, puede generar grandes volúmenes de datos y su análisis suele realizarse de forma *offline*. La Figura 2.2 muestra un ejemplo de este método.

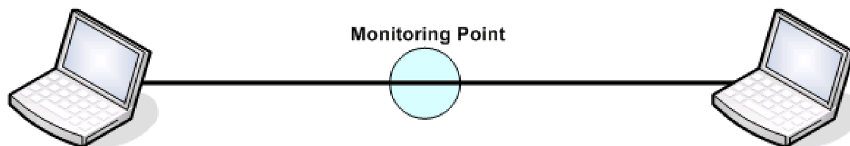


Figura 2.2: Monitoreo pasivo [5].

2.2. Dispositivos de red programables

La capacidad del *software* o del *hardware* para ejecutar un algoritmo de procesamiento definido externamente es la característica que distingue a entidades programables de entidades flexibles (o configurables). Estas últimas permiten cambiar distintos parámetros del algoritmo definido internamente, que no se modifica. En este sentido, el término programabilidad de la red se refiere a la capacidad de definir el algoritmo de procesamiento que se ejecuta en una red y, específicamente, en los nodos individuales de procesamiento, como *switches*, *routers*, balanceadores de carga, entre otros [22].

Los dispositivos de red tradicionales, como *routers* y *switches*, integran el plano de control y el plano de datos en un mismo equipo. El plano de control

define políticas de procesamiento de paquetes (por ejemplo, rutas o modificaciones de encabezados), mientras que el plano de datos se encarga de ejecutar estas políticas, reenviando los paquetes según lo indicado [26]. Aunque los usuarios pueden configurar ciertos parámetros mediante interfaces de administración (CLI, web o APIs), los algoritmos internos solo pueden ser modificados por los fabricantes, lo que limita la innovación y la incorporación rápida de nuevas funciones [4].

La programabilidad de la red permite definir y modificar los algoritmos tanto del plano de control como del plano de datos, permitiendo a usuarios y fabricantes adaptar los dispositivos a necesidades específicas sin depender de los diseñadores originales del *hardware*. Esto posibilita la creación de redes más flexibles, eficientes y adaptadas a distintos casos de uso, reduciendo costos y tiempos de desarrollo sin comprometer el rendimiento [4].

Mientras que el plano de gestión se volvió programable en la década de 1980, la programabilidad en el plano de control no se logró hasta finales de la década del 2000 y principios de la del 2010. La programabilidad del plano de datos (en ASICs, *application-specific integrated circuits*) es más reciente aún [22].

La red definida por software (SDN por sus siglas en inglés *Software-defined networking*) fue el primer intento de hacer programable el plano de control. Algunos fabricantes comenzaron a permitir que los usuarios reemplazaran los algoritmos internos de control, aunque el plano de datos seguía siendo cerrado y dependiente del proveedor [22]. La figura 2.3 muestra la diferencia entre SDN y una red tradicional.

La programabilidad en el plano de datos se aborda en la siguiente sección.

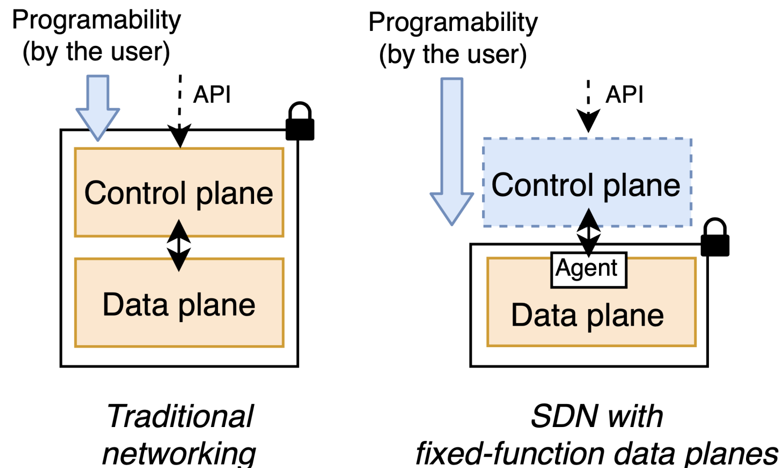


Figura 2.3: Diferencia entre redes tradicionales y SDN con plano de datos cerrado [22].

2.3. Programabilidad del plano de datos

La programabilidad del plano de datos supone que los algoritmos encargados del procesamiento de paquetes puedan ser definidos y modificados por los propios usuarios. Históricamente, esta capacidad no representaba un problema, ya que el procesamiento de paquetes se realizaba en una *CPU* de propósito general. Sin embargo, con el crecimiento de las velocidades de enlace y el aumento del volumen de tráfico, las *CPU* dejaron de ser suficientes para mantener el ritmo del procesamiento a nivel de línea. Esto llevó al desarrollo de ASICs dedicados al reenvío de paquetes, optimizados para rendimiento, pero mucho más rígidos en cuanto a su capacidad de ser modificados o re-programados. De esta forma, la pérdida de flexibilidad en el plano de datos emergió como un nuevo desafío en el diseño de redes modernas [22].

Los algoritmos del plano de datos procesan todos los paquetes que atraviesan un sistema de telecomunicaciones, definiendo su funcionalidad, rendimiento y escalabilidad. Intentar ejecutar estas tareas en el plano de control suele degradar significativamente el desempeño. La programabilidad del plano de datos permite a usuarios y fabricantes crear equipos y redes personalizadas sin comprometer velocidad ni eficiencia, adaptando los algoritmos a necesidades específicas o a nuevos planos de control y aplicaciones SDN [22].

Los algoritmos del plano de datos pueden expresarse, y a menudo se expresan, mediante lenguajes de programación convencionales. Sin embargo, su traducción directa a *hardware* especializado, como los ASICs de alta velocidad, no resulta eficiente. Para abordar este problema, se han propuesto distintos modelos de plano de datos que abstraen el funcionamiento del *hardware*. Sobre estos modelos se construyen lenguajes de programación específicos que permiten describir de forma abstracta los algoritmos de procesamiento, los cuales luego se compilan para ejecutarse en dispositivos concretos que soportan dicho modelo. Entre estos modelos destacan las abstracciones basadas en grafos de flujo de datos (o *Data Flow Graph Abstractions*) y la *Protocol Independent Switching Architecture* (PISA), siendo esta última la base sobre la que se desarrolla el lenguaje P4 [22].

2.3.1. Protocol-Independent Switching Architecture (PISA)

La arquitectura PISA está basada en el concepto de un *pipeline match-action* programable que sea compatible con el *hardware* moderno de *switching*. En PISA, dicho *pipeline match-action* se encuentra entre un *parser* programable y un *deparser* programable [22]. La figura 2.4 muestra la arquitectura de PISA.

El *parser* programable permite declarar encabezados arbitrarios y una maquina de estados finita que define el orden de esos encabezados dentro de los paquetes. El *deparser* se encarga de la acción opuesta, es decir, se encarga de definir en que orden se vuelven a serializar los paquetes [22].

El *pipeline* de *match-action* está compuesto por múltiples unidades *match-action*. Cada una de estas unidades contiene una o más *Match-Action Tables*

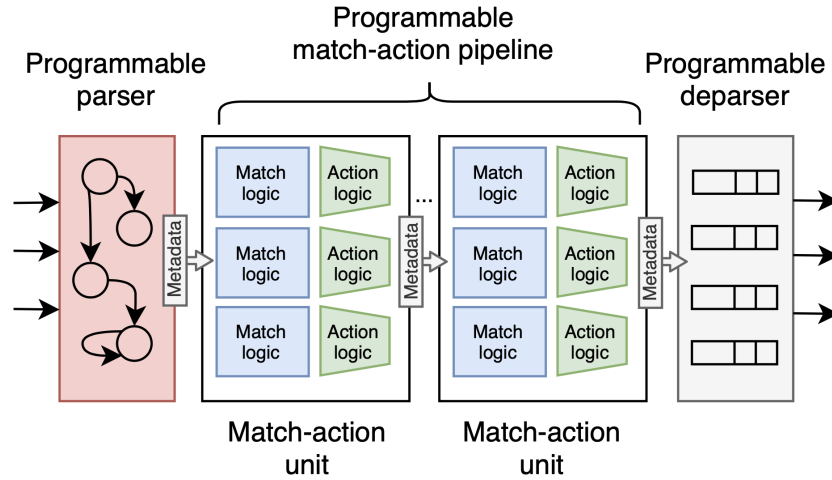


Figura 2.4: Protocol-Independent Switching Architectura (PISA) [22].

(MATs), que se encargan de *matchear* paquetes y ejecutar acciones específicas asociadas a cada *match*, utilizando los datos de acción correspondientes. La mayor parte del procesamiento de paquetes se define en las MATs, que combinan la lógica de match con memoria (*SRAM* o *TCAM*) para almacenar claves y acciones. Las operaciones resultantes, como modificaciones de encabezados o cálculos, se ejecutan mediante ALUs (Arithmetic Logic Units) y pueden complementarse con objetos con estado (contadores, medidores o registros). El plano de control gestiona las MATs escribiendo sus entradas y ajustando dinámicamente el comportamiento del *pipeline* [22].

En PISA, los metadatos de los paquetes pueden ser separados en: *headers*, que corresponden a los encabezados de los protocolos de red y suelen ser extraídos en el *parser* y emitidos en el *deparser*; *intrinsic metadata*, que está relacionada con los componentes no programables (*fixed-function*); y *user-defined metadata* (o simplemente metadata), el análogo a variables en otros lenguajes de programación, permiten guardar información temporalmente en los paquetes para que sea usada a lo largo del *pipeline*.

PISA solamente procesa los metadatos que viajan desde el *parser* hasta el *deparser*, pero no procesa el *payload* del paquete, que viaja por separado.
Todos los metadatos, ya sean *intrinsic metadata*, *user-defined metadata*, o *headers* son temporales, es decir, son descartados cuando el paquete abandona el *pipeline*, ya sea por que es enviado por un puerto de egreso o es descartado.

PISA ofrece un modelo abstracto que puede adaptarse para crear distintas arquitecturas concretas. Permite definir *pipelines* con diversas combinaciones de componentes programables, por ejemplo, con uno o más *parsers* y *deparsers*, o con múltiples *match-action pipelines* intermedios, además de incluir compo-

nentes especializados para tareas avanzadas como cálculos de *hash* o *checksum*. Junto a estos elementos programables, las arquitecturas de *switches* suelen incorporar componentes fijos configurables, como bloques de puertos de entrada y salida, *replication engines* para *multicasting* o *mirroring*, y gestores de tráfico encargados del *buffering*, encolado y planificación de paquetes. Los componentes fijos interactúan con los programables mediante *intrinsic metadata* [22].

La figura 2.5 muestra una arquitectura típica de *switch* basada en PISA, compuesta por *pipelines* programables de *ingress* y *egress*, junto con tres componentes de función fija: un bloque de *ingress*, otro de *egress*, y un *replication engine* con un gestor de tráfico intermedio.

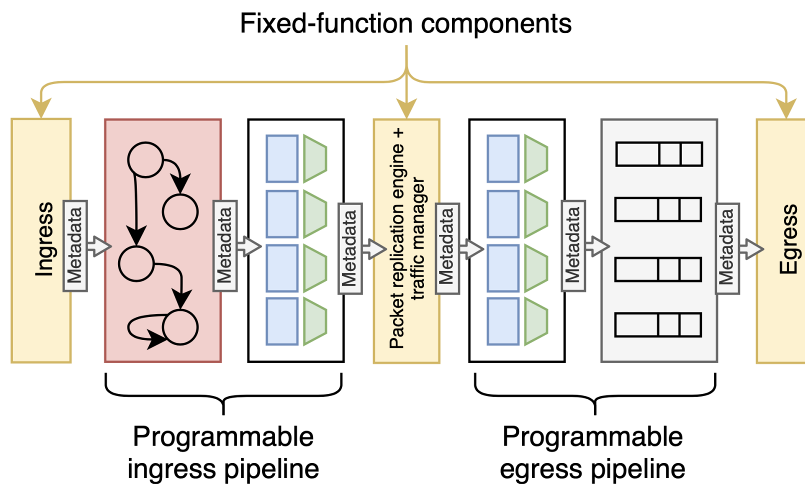


Figura 2.5: Ejemplo de *switch* basado en PISA [22].

P4 [3] es el lenguaje de programación de propósito específico (*Domain Specific Language*, DSL) más utilizado para describir algoritmos del plano de datos en arquitecturas basadas en PISA. Presentado en 2013 y formalizado en 2014, ha sido desarrollado y estandarizado por el P4 *Language Consortium*, actualmente parte de la *Open Networking Foundation* (ONF). La primera versión estándar fue $P4_{14}$ [35], mientras que la versión actual es $P4_{16}$ [33], introducida en 2016. Además de P4, existen otros lenguajes orientados al plano de datos, como *FAST* [29], *OpenState* [2], Domino [44], *FlowBlaze* [37], *Protocol-Oblivious Forwarding* [45] y *NetKAT* [1].

2.3.2. El lenguaje P4

P4 es un lenguaje diseñado para describir cómo los paquetes son procesados en el plano de datos de dispositivos de reenvío programables, como *switches*, *routers*, *NICs* o dispositivos de red tanto en *hardware* como en software. Su

nombre proviene del trabajo original "*Programming Protocol-Independent Packet Processors*" [3]. Aunque inicialmente fue concebido para programar *switches*, su alcance se ha ampliado para abarcar una amplia gama de dispositivos, denominados de forma genérica *targets* [33].

Un *switch* programable en P4 se diferencia de uno tradicional en dos aspectos fundamentales. Primero, su plano de datos no tiene una funcionalidad predefinida, sino que esta se especifica mediante un programa P4 que se carga durante la inicialización. Esto implica que el dispositivo no posee conocimiento incorporado de protocolos de red existentes. Segundo, aunque el plano de control se comunica con el de datos por los mismos canales que en un dispositivo convencional, las tablas y objetos del plano de datos ya no son fijos, sino que son definidos por el programa P4. El compilador de P4 genera automáticamente la API que el plano de control utiliza para interactuar con el plano de datos. De esta forma, P4 es independiente del protocolo, pero permite describir una amplia variedad de comportamientos y protocolos en el plano de datos [33].

Las principales abstracciones que provee P4 son:

- ***Header types*:** Describen el formato (conjunto de campos y sus tamaños) de cada header dentro de un paquete.

- ***Parsers*:** Definen las secuencias permitidas de encabezados en los paquetes recibidos, cómo identificar esas secuencias y qué campos extraer.

- ***Tables*:** Asociaciones entre claves definidas por el usuario y acciones. Generalizan las tablas tradicionales de *switches* y pueden implementarse para *routing*, *flow lookup*, *access-control lists*, y decisiones complejas multivariable.

- ***Actions*:** Fragmentos de código que manipulan campos de *headers* y metadatos de paquetes. Pueden incluir datos suministrados por el plano de control en tiempo de ejecución.

- ***Match-action units***: Secuencia de operaciones que incluyen:
  - Construcción de claves a partir de campos de paquetes o metadata calculada.
  - Búsqueda en tablas usando la clave construida para seleccionar la acción a ejecutar.
  - Ejecución de la acción seleccionada.

- ***Control flow*:** Programa imperativo que describe el procesamiento de paquetes en un *target*, incluyendo la secuencia dependiente de datos de invocaciones de *match-action units*. También permite el *deparsing* (reensamblado de paquetes).

- ***Extern objects*:** Elementos específicos de la arquitectura accesibles mediante APIs, pero cuyo comportamiento interno está fijo (ej. unidades de *checksum*) y no son programables con P4.

- ***User-defined metadata*:** Estructuras de datos definidas por el usuario asociadas a cada paquete.
- ***Intrinsic metadata*:** Metadata proporcionada por la arquitectura asociada a cada paquete, por ejemplo, el puerto de entrada donde se recibió un paquete.

Los fabricantes de *targets* proporcionan el *hardware* o software, la definición de arquitectura y un compilador P4 específico para ese *target*. Los programadores P4 escriben programas para una arquitectura concreta, que define los componentes programables y sus interfaces del plano de datos. La compilación de los programas genera dos elementos: la configuración del plano de datos que implementa la lógica de reenvío y una API para que el plano de control gestione el estado de los objetos del plano de datos. P4 es un lenguaje de dominio específico diseñado para ser ejecutable en múltiples plataformas, incluyendo NICs programables, FPGAs (*Field-Programmable Gate Array*), *switches* por software y ASICs, lo que limita su sintaxis a construcciones eficientes en todos estos entornos.

La arquitectura P4 define los bloques programables (como el *parser*, el flujo de control de ingreso, el flujo de control de egreso y el *deparser*) y sus interfaces dentro del plano de datos. Puede entenderse como un contrato entre el programa y el *target*. Por ello, cada fabricante debe proporcionar tanto un compilador P4 como una definición de arquitectura correspondiente para su *target*. Además, una arquitectura no necesita exponer toda la superficie programable del plano de datos; un fabricante incluso puede ofrecer múltiples definiciones para el mismo *hardware*, cada una con distintas capacidades [33].

P4 ofrece varias ventajas sobre otros sistemas de procesamiento de paquetes. Proporciona flexibilidad al permitir que muchas políticas de reenvío se expresen como programas en lugar de *engines* de reenvío fijos. Su expresividad permite definir algoritmos sofisticados independientes del *hardware* mediante operaciones generales y búsquedas en tablas, lo que facilita la portabilidad entre distintos *targets*. Además, permite la gestión abstracta de recursos de almacenamiento, la reutilización y verificación de programas mediante buenas prácticas de ingeniería de software, y el uso de librerías de componentes que encapsulan funciones específicas del *hardware*. P4 también desacopla la evolución del *hardware* y software a través del uso de arquitecturas abstractas [33].

BMv2

BMv2 es el *switch* de referencia en software para programas P4. Ejecuta sobre *CPU* de propósito general e interpreta un artefacto *JSON* generado por el compilador *p4c* a partir del programa fuente, reproduciendo el comportamiento del plano de datos especificado por el desarrollador [7, 8]. Su propósito central no es el despliegue en producción, sino desarrollar, probar y depurar tanto planos de datos como el software de plano de control asociado.

Este carácter enteramente *software* lo convierte en una herramienta idónea para evaluar implementaciones de P4: permite iterar rápidamente sobre *parsers*,

tablas y acciones; instrumentar con precisión (por ejemplo, mediante trazas o capturas *pcap*); y ejecutar pruebas reproducibles en entornos de laboratorio. De esta forma, BMv2 ofrece un entorno controlado para validar la correctitud funcional de un programa antes de su portado a *targets* de *hardware*.

Arquitecturas compatibles. El lenguaje P4 no define una arquitectura fija, sino que permite describir distintas configuraciones de plano de datos según el tipo de dispositivo programable. En este contexto, una arquitectura especifica qué bloques son programables (por ejemplo, *parser*, flujo de ingreso y egreso, *deparser*), qué metadatos expone y de qué manera el plano de control puede interactuar con ellos. Así, la arquitectura actúa como un contrato entre el programa P4 y el *target* sobre el que será ejecutado.

El compilador oficial *p4c* [8] proporciona varias arquitecturas de referencia. La más extendida es *v1model*, diseñada para mantener compatibilidad con el modelo de *switch* definido en P4 [33].

BMv2 implementa `v1model` en sus variantes `simple_switch` y `simple_switch_grpc`, lo que lo convierte en la plataforma más utilizada para validar programas P4 sobre una arquitectura de conmutador genérica. Además, existe una implementación parcial de la PSA [34], que busca ofrecer una interfaz más moderna y portable entre diferentes tipos de dispositivos (por ejemplo, *switches*, *routers* y *SmartNICs*). Sin embargo, al momento de esta redacción, la versión *psa_switch* aún no alcanza el mismo nivel de madurez ni soporte funcional que *v1model* [7].

Metadatos y análisis. BMv2 expone tanto metadatos estándar (`standard_metadata_t`, por ejemplo `ingress_port`, `packet_length` o `egress_spec`) como metadatos intrínsecos específicos del modelo de software. Estos incluyen marcas de tiempo de ingreso y egreso (`ingress_global_timestamp`, `egress_global_timestamp`) y métricas de encolamiento (`enq_qdepth`, `deq_timedelta`, `deq_qdepth`) útiles para medir latencias y ocupación de colas [7]. Dichos campos resultan especialmente valiosos al evaluar mecanismos de INT o al correlacionar tiempos de tránsito en *pipelines* programables [10].

Alcance y limitaciones. Al ser un plano de datos de software, BMv2 no reproduce el rendimiento de un ASIC o *SmartNIC*. Sus resultados son adecuados para verificación funcional y de control, pero no para mediciones precisas de rendimiento o latencia absoluta. Además, las marcas de tiempo internas dependen del reloj local del proceso, sin sincronización PTP (Precision Time Protocol), por lo que las comparaciones temporales entre instancias requieren precaución [9].

2.4. In-band Network Telemetry (INT)

La información presentada en esta sección fue extraída de la versión 2.1 de la especificación del plano de datos de INT [36].

INT es un *framework* que permite recolectar y reportar el estado de la red directamente desde el plano de datos, sin intervención del plano de control. En este modelo, los paquetes pueden incluir campos en encabezados que actúan como instrucciones de telemetría, interpretadas por los dispositivos de red.
Las fuentes de tráfico INT (como aplicaciones, *stacks* de red, hipervisores, *NICs* o *switches* de borde) pueden insertar estas instrucciones dentro de los paquetes de datos, en copias clonadas de ellos o en paquetes de sondeo especiales.

Las instrucciones indican qué información de estado debe recolectar cada dispositivo que implementa INT. Este estado puede exportarse directamente al sistema de monitoreo o escribirse dentro de los propios paquetes a medida que atraviesan la red. Finalmente, los receptores de tráfico INT (INT *Sinks*) extraen y reportan los resultados recopilados, permitiendo observar el estado real del plano de datos que los paquetes experimentaron durante su recorrido.

2.4.1. Arquitectura general de INT

En INT, ciertos nodos de la red, denominados INT *Nodes*, son capaces de insertar, modificar o eliminar información de telemetría dentro de los paquetes. Cada paquete con telemetría contiene un INT *Header*, que puede incluir instrucciones sobre qué información recolectar (por ejemplo, identificadores, estado de colas, timestamps, etc.) y, dependiendo del modo operativo, también los datos recopilados.

Los roles principales dentro de una arquitectura INT son los siguientes:

- **INT *Source*:** nodo que inicia el proceso de telemetría. Inserta el encabezado INT en los paquetes seleccionados según una *Flow Watchlist* (una tabla de coincidencia en el plano de datos que identifica los flujos a monitorear).

- **INT *Transit Hop*:** nodo intermedio que ejecuta las instrucciones incluidas en el encabezado INT, recolectando y, según el modo, insertando metadatos adicionales o exportándolos directamente al sistema de monitoreo.

- **INT *Sink*:** nodo receptor que extrae el encabezado INT y los metadatos acumulados, y opcionalmente envía esta información al sistema de monitoreo o la utiliza localmente para diagnóstico o control.

- ***Monitoring System*:** entidad encargada de centralizar y analizar la información recolectada por los distintos nodos, pudiendo estar físicamente distribuida pero lógicamente centralizada.

Los dispositivos que cumplen alguno de estos roles conforman un INT *Domain*, es decir, un conjunto de nodos interconectados bajo una misma administración y configurados de forma coherente para garantizar la interoperabilidad. En los bordes del dominio se recomienda desplegar capacidades de INT *Sink* para evitar la filtración de información de telemetría hacia el exterior.

De manera general, el funcionamiento de INT sigue un ciclo:

1. El INT *Source* marca los paquetes y define las instrucciones de recolección.

2. Los INT *Transit Hops* añaden los metadatos correspondientes conforme las instrucciones.

3. El INT *Sink* elimina el encabezado INT, extrae la información recolectada y la reporta al sistema de monitoreo.

2.4.2. Modos de operación

Los distintos modos de operación de INT se clasifican según el grado de modificación que se aplica a los paquetes, es decir, que información se inserta o se exporta durante su tránsito por la red. La Figura 2.6 resume visualmente estos modos, mientras que la Tabla 2.1 presenta una comparación detallada.

Modo	**Descripción**	**Modificación del paquete**
INT-XD (*eXport Data*)	Los nodos INT exportan directamente los metadatos desde el plano de datos hacia el sistema de monitoreo, siguiendo las instrucciones de las *Flow Watchlists*. No se modifica el paquete original. También se conoce como *Postcard Mode*.	Ninguna
INT-MX (*eMbed Instructions*)	El nodo INT *source* inserta únicamente las instrucciones de recolección. Los nodos intermedios y el destino generan y envían los metadatos al sistema de monitoreo sin modificar el tamaño del paquete, ya que los metadatos no se embeben en él.	Baja (solo instrucciones)
INT-MD (*eMbed Data*)	Modo clásico de INT. El nodo fuente agrega las instrucciones, los nodos de tránsito embeben sus metadatos dentro del paquete, y el nodo destino extrae y elimina esta información antes de reenviarlo. Reduce la carga del sistema de monitoreo al agregar información a lo largo del camino.	Alta (instrucciones + metadatos)

Tabla 2.1: Comparación de los modos de operación de INT.

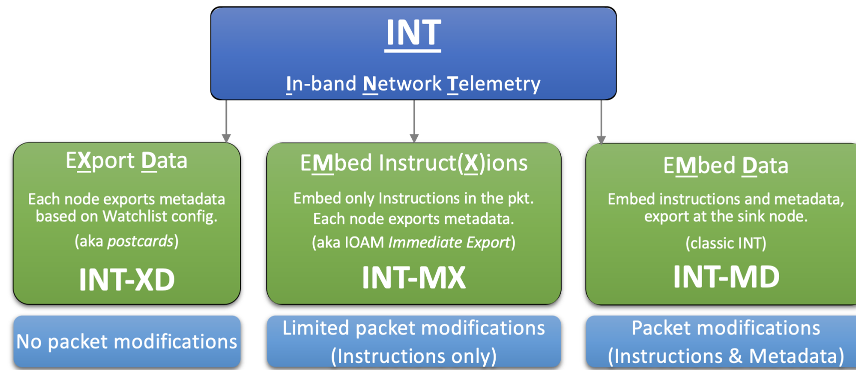


Figura 2.6: Modos de operación INT [36].

INT aplicado a tráfico sintético

Además de los paquetes de datos reales, INT puede aplicarse a tráfico sintético o de prueba. En este caso, el nodo *source* genera paquetes marcados con información INT, ya sea clonando tráfico existente o creando sondas específicas. Los nodos de tránsito los procesan de la misma forma que el tráfico normal, mientras que el nodo destino puede descartar los paquetes tras extraer los metadatos recolectados.

El encabezado INT incluye un bit de control (bit D) que permite marcar los paquetes que deben ser descartados al llegar al nodo destino. Este mecanismo resulta útil para pruebas de diagnóstico, análisis de rendimiento o verificación de rutas, ya que posibilita la observación del comportamiento de la red sin afectar el tráfico de producción.

2.4.3. ¿Qué se monitorea?

En teoría, INT permite definir y recolectar cualquier tipo de información interna de los dispositivos de red. En la práctica, sin embargo, resulta útil establecer un conjunto básico de metadatos que puedan ser soportados de manera amplia por distintos dispositivos. Este conjunto se presenta en la especificación de INT [36] como una base común, con la expectativa de ampliarse en futuras versiones. El significado exacto de cada metadato, por ejemplo, las unidades de tiempo en los timestamps o la definición precisa de latencia por salto y ocupación de cola, puede variar entre dispositivos debido a diferencias de arquitectura, recursos o funcionalidades. Por ello, la especificación no estandariza estas definiciones, sino que asume que su interpretación se comunica fuera de banda (*out of band*) entre los dispositivos y las entidades que analizan los datos recolectados.

Información a nivel de dispositivo

Node id Identificador único (dentro del INT *domain*) de un nodo INT. Se suela asignar administrativamente.

Información de ingreso (*Ingress Information*)

Identificador de interfaz de ingreso Representa la interfaz por la cual un paquete INT fue recibido en el dispositivo. En la práctica, un paquete puede llegar a través de una pila de interfaces anidadas, por ejemplo, un puerto físico que pertenece a un grupo de agregación de enlaces, que a su vez forma parte de una interfaz virtual de Capa 3 o incluso un túnel. Aunque en teoría podría registrarse toda esta jerarquía, la especificación INT permite monitorear hasta dos niveles de identificadores de interfaz de ingreso:

- **Primer nivel:** normalmente se utiliza para identificar el puerto físico de recepción. Se representa en un campo de 16 bits.
- **Segundo nivel:** se reserva un campo de 32 bits, destinado típicamente a identificar una interfaz lógica.

El significado exacto de estos identificadores puede variar entre dispositivos, y cada nodo INT decide qué tipo de interfaz reportar en cada nivel.

Ingress timestamp Corresponde al tiempo local del dispositivo en el momento en que el paquete INT fue recibido por el puerto físico o lógico de ingreso.

Información de egreso (*Egress Information*)

Identificador de interfaz de egreso Indica la interfaz por la cual el paquete INT fue transmitido desde el dispositivo. Similar al caso de ingreso, un paquete puede salir a través de una pila de interfaces. La especificación permite monitorear hasta dos niveles de identificadores de egreso:

- **Primer nivel:** normalmente se utiliza para identificar el puerto físico de salida. Se representa en un campo de 16 bits.
- **Segundo nivel:** se reserva un campo de 32 bits, destinado típicamente a identificar una interfaz lógica de salida.

El significado exacto de estos identificadores puede variar entre dispositivos, y cada nodo INT decide qué tipo de interfaz reportar en cada nivel.

Egress timestamp Corresponde al tiempo local del dispositivo en el momento en que el paquete INT fue procesado por el puerto físico o lógico de egreso.

Latencia por salto (*Hop latency*) Representa el tiempo total que el paquete INT tardó en atravesar el dispositivo, es decir, el tiempo entre su recepción en el puerto de ingreso y su transmisión en el puerto de egreso.

Utilización del enlace de egreso (*Egress interface TX Link utilization*) Indica la utilización del enlace de salida en el momento en que el paquete fue transmitido. Los dispositivos pueden estimar esta métrica mediante diferentes métodos. La especificación INT no impone un método particular, dejando esta decisión a los fabricantes.

Ocupación de cola (*Queue occupancy*) Mide la cantidad de tráfico en la cola de transmisión (en bytes, celdas o paquetes) que el paquete INT "observa" mientras es reenviado. Este valor se almacena en un campo de 4 bytes, cuyo formato y unidades dependen de la implementación.

Ocupación de *buffer* (*Buffer occupancy*) Indica la acumulación de tráfico en el buffer compartido entre múltiples colas dentro del dispositivo, también expresada en bytes, celdas o paquetes. Al igual que la ocupación de cola, el formato de este metadato es dependiente del fabricante, y se almacena en un campo de 4 bytes.

2.4.4. *Headers* INT

La especificación INT define un conjunto de *headers* (INT *Headers*) que determinan dónde y cómo se almacenan las instrucciones de telemetría y los metadatos dentro de los paquetes. Estos encabezados son relevantes en los modos *INT-MX* y *INT-MD*, donde las instrucciones (y en el caso de *INT-MD*, también la pila de metadatos) son insertadas directamente en los paquetes de datos.

Tipos de *headers* INT

Existen tres tipos de *headers* INT, resumidos en la Tabla 2.2.

Tipo	¿Quién lo procesa?	Descripción
MD-type (INT Header Type 1)	Nodos intermedios	Los nodos intermedios deben procesarlo. El formato del encabezado se detalla en la Sección 2.4.6.
Destination-type (INT Header Type 2)	Solo el INT *Sink*	Los nodos intermedios deben ignorarlo. Permite comunicación *Edge-to-Edge* entre el INT *Source* y el INT *Sink*.
MX-type (INT Header Type 3)	Nodos intermedios	Los nodos intermedios deben procesarlo y generar reportes para el sistema de monitoreo del modo que se indica.

Tabla 2.2: Tipos de *headers* INT.

Operaciones por salto (*Per-Hop Header Operations*)

Nodo fuente (*INT Source Node*) El INT *Source* es el responsable de insertar el encabezado INT (ya sea *MD* o *MX*) en el paquete.

- En modo *INT-MD*, el nodo fuente también agrega su propio bloque de metadatos inmediatamente después del encabezado.

- Para prevenir el desbordamiento del espacio reservado al encabezado el nodo fuente debe establecer el campo *Remaining Hop Count* con un valor que limite la cantidad de nodos que pueden insertar metadatos.

Nodo de tránsito (*INT Transit Hop Node*) Cada nodo intermedio procesa los encabezados INT de manera distinta según el modo de operación:

- En *INT-MD*, el nodo extiende dinámicamente el encabezado para incluir su propio conjunto de metadatos, y decrementa el campo *Remaining Hop Count* para reflejar su participación.

- En *INT-MX*, el nodo sigue las instrucciones del encabezado, recolecta los metadatos correspondientes al dispositivo y genera un reporte de telemetría (*Telemetry Report*) hacia el sistema de monitoreo.

- Los nodos de tránsito pueden modificar el campo *DS Flags*, pero no deben alterar los campos *Hop ML*, *Instruction Bitmap*, *Domain Specific ID* ni *DS Instruction*, que permanecen fijos durante el tránsito.

Configuración de MTU

En los modos *INT-MD* e *INT-MX*, la inserción de encabezados INT puede hacer que el tamaño del paquete exceda la MTU del enlace de egreso, especialmente en *INT-MD*, donde cada salto inserta sus metadatos. Para evitarlo, la especificación [36] recomienda aumentar la MTU de los enlaces dentro del dominio INT, reservando un margen acorde al tamaño del encabezado y de los metadatos esperados por salto. Alternativamente, los nodos pueden participar en *PMTUD* (*Path MTU Discovery*) enviando mensajes ICMP con valores conservadores. Si un nodo no puede insertar todos los metadatos sin exceder la MTU, debe ya sea omitir la inserción y marcar el bit M ß(MTU superado), o reportar y eliminar la pila de metadatos acumulada antes de reenviar el paquete. La especificación también establece que los nodos INT no deben fragmentar paquetes para incorporar información adicional.

INT sobre cualquier encapsulación

La especificación de INT [36] no define una ubicación fija para los encabezados INT dentro de los paquetes. En su lugar, establece que estos encabezados pueden insertarse como opciones o *payloads* en distintos tipos de encapsulación.

Esta flexibilidad permite que INT se adapte a múltiples entornos y protocolos de red. En este trabajo se utiliza la encapsulación de INT sobre TCP/UDP, la cual se detalla en la siguiente sección.

2.4.5. INT sobre TCP/UDP

Los metadatos INT pueden insertarse inmediatamente después de los encabezados de capa 4 (TCP o UDP). Este método asume que los dispositivos que no soportan INT no inspeccionan más allá de la capa de transporte, o que pueden saltar la pila INT utilizando el campo *Length* del encabezado *int shim*.

En el caso de TCP, si hay TCP *options*, la pila INT puede ubicarse antes o después de ellas, siempre que la elección sea consistente dentro del dominio INT.

Se requiere que algún campo del encabezado *Ethernet*, *IP* o TCP/UDP indique la presencia del encabezado INT tras la capa de transporte, garantizando que los nodos INT puedan reconocer e interpretar correctamente la telemetría embebida. Para eso hay tres opciones:

1. Puerto de destino UDP: la presencia del encabezado INT puede señalarse mediante el uso de un puerto de destino específico (INT_TBD), que será asignado por *IANA*. Este mecanismo permite a los nodos de la red identificar fácilmente que un paquete contiene información INT. Esta opción contempla dos casos:

   a) Paquetes con encabezado UDP original: Si el paquete ya incluye un encabezado UDP el encabezado INT se inserta después de este, y el puerto de destino UDP se reemplaza por INT_TBD. El valor original del puerto se guarda en el encabezado *shim*, para que el nodo INT *Sink* pueda restaurarlo al eliminar la pila INT.

   b) Inserción de un nuevo encabezado UDP: En los casos en que no exista un encabezado UDP, se puede insertar uno entre *IP* y la capa *L4* original, cambiando el protocolo *IP* a UDP (17). En este encabezado nuevo, el puerto de destino se configura como INT_TBD, mientras que el valor original del protocolo *IP* se conserva dentro del encabezado *shim* para su posterior restauración. El puerto de origen se recomienda calcularlo a partir de un *hash* del flujo original (por ejemplo, la 5-tupla *IP*) para mantener el equilibrio de carga en esquemas *ECMP*/*LAG*.

   El campo NPT (*Next Protocol Type*) del encabezado *shim* indica cuál de los dos casos aplica. En la etapa final, el INT *Sink* restaura los valores originales (ya sea el puerto o el protocolo *IP*) y elimina los encabezados INT y el UDP insertado, devolviendo el paquete a su forma inicial.

2. **Uso del campo DSCP (*IPv4*) o *Traffic Class* (*IPv6*)**: Se puede utilizar un valor o un bit del campo DSCP/*Traffic Class*. Cuando el nodo

fuente inserta el encabezado INT, marca dicho campo con un valor reservado, y opcionalmente almacena el valor original dentro del encabezado INT para que el *Sink* pueda restaurarlo.

3. **Campos Probe Marker**: Si no es posible reservar valores o bits en el campo DSCP, puede utilizarse un mecanismo alternativo mediante probe markers. Este método consiste en insertar un valor fijo de 64 bits inmediatamente después del encabezado TCP/UDP, para señalar la presencia del encabezado INT.

   Estos valores se interpretan como enteros sin signo en *network byte order* y actúan como identificadores únicos del tráfico INT. Aunque la probabilidad de colisión con tráfico regular es baja, no puede descartarse completamente; por ello, se recomienda que las implementaciones también verifiquen los números de puerto TCP/UDP para confirmar la validez del marcador y reducir aún más el riesgo de conflicto.

Dentro de un mismo dominio INT, puede utilizarse cualquiera de los mecanismos descritos (UDP *port*, DSCP/*Traffic Class* o *Probe Marker*), siempre que todos los nodos *source*, tránsito y *Sink* adopten el mismo método y sean capaces de identificar correctamente la presencia y ubicación de los encabezados INT.

Estos mecanismos no están diseñados para interoperar entre sí: combinar más de uno en un mismo entorno (por ejemplo, usar simultáneamente DSCP y *Probe Marker*) podría provocar interpretaciones erróneas. Por ejemplo, un nodo que reconozca solo la marca DSCP podría confundir los primeros bytes del *Probe Marker* con un encabezado *shim* de INT.

INT Shim Header

Es el primero de los *headers* INT. El *header* INT metadata y el *stack* de metadata de los nodos se encuentra entre el *header shim* y el *payload* TCP/UDP. La figura 2.7 muestra el *header shim* para TCP/UDP.

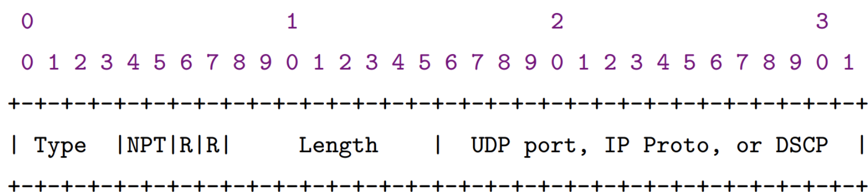


Figura 2.7: INT *shim header* para TCP/UDP [36].

- **Type (4b)**: Indica el tipo de INT *header* que sigue al *header shim*. Las opciones posibles son las descritas en la sección 2.4.4

- **NPT (Next Protocol Type, 2b)**: El campo NPT solo tiene relevancia cuando el número de puerto de destino UDP (INT_TBD) se usa para indicar la presencia de un encabezado INT. En cualquier otro caso, su valor debe ser 0.

  Cuando se utiliza el puerto de destino INT_TBD, este campo puede tomar dos valores posibles:

  - **uno (1)**: indica que el *payload* UDP original sigue a la pila INT. En este caso, los últimos dos bytes del encabezado *shim* contienen el puerto de destino UDP original.
  - **dos (2)**: indica que tras la pila INT se encuentra otro encabezado de capa 4 (*L4*) original, y el último byte del encabezado *shim* almacena el valor del protocolo *IP* correspondiente a esa capa *L4*.

- **Length (8b)**: Este campo indica la longitud total del encabezado de metadatos INT y de la pila INT, expresada en palabras de 4 bytes. Desde la versión INT 2.0, no incluye la longitud del encabezado *shim* (1 palabra). Los dispositivos que no soportan INT pueden usar este valor para saltar los encabezados INT sin procesarlos.

- **UDP port, *IP* proto, or DSCP (16b)**: El contenido de este campo depende del valor del campo NPT.

  - **NPT=0**: el primer byte y los últimos dos bits están reservados. Los primeros 6 bits del segundo byte pueden cargar, opcionalmente, el valor original de DSCP.
  - **NPT=1**: contiene el valor original del puerto de destino UDP.
  - **NPT=2**: el primer byte está reservado y el segundo contiene el valor original del protocolo *IP* asociado a la capa 4.

Los demás bits del encabezado *shim* están reservados para uso futuro, se transmiten en cero y deben ser ignorados al recibir el paquete.

2.4.6. Formato de *INT-MD* Metadata Header

El *INT-MD* metadata *header* es el *header* que sucede al INT *shim header*. En la figura 2.8 se puede observar el *INT-MD* metadata *header* sucedido por el *stack* de metadatos INT en sí.

El *INT-MD* metadata *header* tiene un largo de 12 bytes y consiste en los siguientes campos:

- **Ver (4b):** : versión del INT metadata *header* version. La última versión es 2.

- **D (1b):** *Discard*. El nodo *Sink* debe descartar el paquete luego de extraer los metadatos.

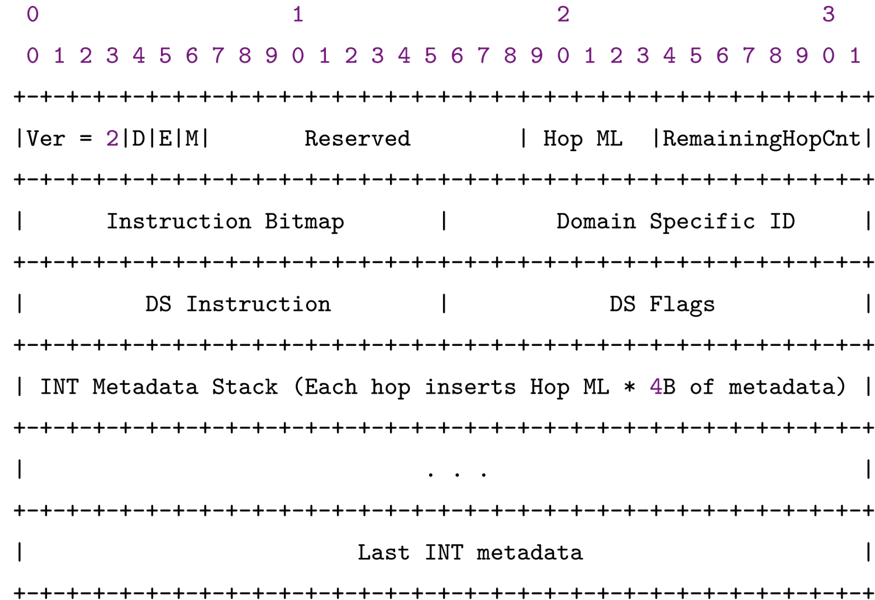


Figura 2.8: *INT-MD* metadata *header* y *stack* de metadatos INT [36].

- **E (1b):** *Max Hop Count* excedido. La *flag* se *setea* cuando un nodo no pueda agregar sus metadatos debido a que el *Remaining Hop Count* llegó a cero. El bit E debe inicializarse en cero por el nodo *source*.

- **M (1b):** MTU excedido. La *flag* se *setea* cuando un nodo no puede insertar todos sus metatados porque de hacerlo se excedería el MTU. El nodo debe *setear* el bit y no agregar ningún metadato. Este indicador sirve para advertir al sistema de monitoreo que uno o más nodos a lo largo del camino no agregaron su metadata INT debido a restricciones de tamaño del paquete. No permite identificar directamente cuál fue el nodo que tuvo el problema.

- **R (12b):** Bits reservados. Deben *setearse* en cero en el nodo source y ser ignorados por el resto nodos.

- **Hop ML (5b):** *Per-hop Metadata Length*. Es el largo de metadatos que insertará cada nodo medido en palabras de 4-bytes. Es *seteado* por el nodo *source* y debe ser respetado por el resto de nodos.

- **Remaining Hop Count (8b):** La cantidad restante de nodos que pueden agregar sus metadatos. El nodo *source* debe inicializar este campo con la cantidad máxima de nodos que pueden agregar metadatos. Todos los nodos (incluyendo el *source*) deben decrementar este valor al *pushear* sus

metadatos. Si un nodo se encuentra el *remaining hop count* en cero, no debe *pushear* sus metadatos, y además debe *setear* la *flag* E.

- **Instruction Bitmap:** Cada bit corresponde a un metadato que debe ser agregado por los nodos INT.

  - **bit0 (MSB)**: Identificador del nodo (*Node ID*)
  - **bit1**: Identificadores de interfaz de entrada y salida de nivel 1 (*Ingress Interface ID* de 16 bits + Egress Interface ID de 16 bits)
  - **bit2**: Latencia del salto (*Hop latency*)
  - **bit3**: Identificador de cola (Queue ID, 8 bits) + Ocupación de la cola (Queue occupancy, 24 bits)
  - **bit4**: Marca de tiempo de ingreso (Ingress timestamp, 8 bytes)
  - **bit5**: Marca de tiempo de egreso (Egress timestamp, 8 bytes)
  - **bit6**: Identificadores de interfaz de entrada y salida de nivel 2 (*Ingress Interface ID* + *Egress Interface ID*, 4 bytes cada uno)
  - **bit7**: Utilización de transmisión de la interfaz de egreso (Egress interface Tx utilization)
  - **bit8**: Identificador del búfer (*Buffer ID*, 8 bits) + Ocupación del búfer (*Buffer occupancy*, 24 bits)
  - **bit15**: Complemento de verificación (*Checksum Complement*)
  - Los bits restantes están reservados.

  Cada bit de instrucción que se activa solicita la inserción de 4 bytes de metadatos en cada salto, excepto los bits 4, 5 y 6, que requieren 8 bytes de metadatos cada uno. La longitud de metadatos por salto (*Hop ML*) se configura en acordemente en el INT *source*.

- ***Domain Specific ID* (16b):** El identificador único del dominio INT. Si el *Domain Specific ID* coincide con algún *Domain ID* conocido por este nodo, entonces se requiere un procesamiento adicional de los *Domain Specific Flags* (*DS Flags*) y de las *Domain Specific Instructions* (*DS Instruction*).

- ***DS Instruction* (16b):** define un mapa de bits de instrucciones específicas para el dominio INT identificado por el *Domain Specific ID*. Cada bit activado indica que deben añadirse metadatos específicos del dominio (*Domain Specific Metadata*) al conjunto de metadatos base antes de insertar el *Checksum Complement*. La cantidad total de metadatos específicos del dominio debe ser un múltiplo de 4 bytes, y su tamaño por salto debe ser coherente con el valor de *Hop ML* configurado por el origen INT.

Cada metadato del *stack* tiene largo 4 bytes u 8 bytes. Cada nodo INT agrega el mismo largo de metadatos, con la excepción del *source*, en caso de que existan metadatos *source-only*. El largo total del *stack* de metadatos es variable debido a que distintos paquetes pueden recorrer distintos caminos y por tanto

distinta cantidad de INT *hops*.

Cada nodo INT de tránsito, así como el nodo *source*, agrega sus propios metadatos según lo indicado por los campos *Instruction Bitmap* y *DS Instruction*, colocándolos inmediatamente después del encabezado de metadatos INT. Los nuevos metadatos se agregan al inicio del conjunto existente, siguiendo una estructura tipo pila, donde los metadatos más recientes quedan arriba. Si un nodo no puede obtener un valor de metadato solicitado, inserta un valor reservado de `0xFF` para indicarlo como inválido.

Si recibe un bit reservado en el *Instruction Bitmap*, el nodo debe rellenar el espacio correspondiente con `0xFFFFFFFF` o no agregar metadatos.

Si un nodo no puede soportar todas las instrucciones o insertar la longitud de metadatos definida, debe insertar el tamaño esperado con valores reservados o no agregar metadatos. Si no agrega metadatos, no debe decrementar el campo *Remaining Hop Count*.

El nodo *Sink* puede agregar sus metadatos locales directamente en la pila INT o incluirlos en el reporte de telemetría. La elección depende de la implementación, pero los sistemas de monitoreo deben reconocer ambos formatos.

A modo de resumen:

- El INT *source* debe *setear* los siguientes campos:
  - *Ver*, *D*, *M*, *Hop ML*, *Remaining Hop Count* e *Instruction Bitmap*.
  - Debe asignar el valor cero a todos los bits reservados.
  - Puede opcionalmente definir los campos específicos del dominio (*Domain-specific fields*).
- Los nodos de tránsito intermedios pueden modificar únicamente los siguientes campos:
  - *E*, *M*, *Remaining Hop Count* y *DS Flags*.
  - No deben alterar los campos *Hop ML*, *Instruction Bitmap*, *Domain Specific ID* ni *DS Instruction* del encabezado *INT-MD*.
- La longitud (en bytes) de la pila de metadatos INT debe ser siempre un múltiplo de (`Hop ML × 4`), más el tamaño de los *Domain Specific Metadata* definidos como *source-only* si fueron añadidos por el origen. La longitud total de la pila puede determinarse restando el tamaño fijo de los encabezados INT (12 bytes) del valor (`shim header length × 4`).

2.5. Conclusiones

En este capítulo se presentaron los fundamentos teóricos que enmarcan y motivan el proyecto. A lo largo de la discusión se mostró cómo la incorporación de dispositivos de red programables -y, en particular, la posibilidad de intervenir directamente en el plano de datos mediante lenguajes como P4- abre nuevas oportunidades para el monitoreo de redes, permitiendo enfoques que van mas allá de las técnicas tradicionales. En ese contexto, se introdujo INT como uno de los *frameworks* que emergen de esa evolución.

El capítulo siguiente describe la plataforma experimental empleada en este trabajo, incluyendo el *hardware*, los entornos de programación y las herramientas de visualización, mientras que el capítulo posterior a ese presenta la implementación desarrollada, que pone en práctica los principios conceptuales introducidos aquí.

Capítulo 3

Plataforma y tecnologías utilizadas

Este capítulo presenta las tecnologías que sustentan la plataforma desarrollada: la *SmartNIC Netronome Agilio CX*, utilizada para implementar el INT *Sink*; el *stack* ELK, empleado para almacenar y visualizar los datos recolectados; y las herramientas DPDK y *MoonGen*, usadas enviar tráfico de red en los experimentos. A diferencia del capítulo anterior, que introdujo los conceptos teóricos necesarios para comprender la programabilidad del plano de datos y el funcionamiento de INT, este capítulo se centra en los componentes tecnológicos concretos utilizados para materializar la plataforma desarrollada.

3.1. *SmartNIC Netronome Agilio CX*

En este trabajo se emplea una *Netronome Agilio CX 2×10 GbE*, perteneciente a la línea *Agilio CX* descrita en la documentación oficial [38]. Las *SmartNICs* de la familia *Agilio* están basadas en los procesadores programables *Netronome Network Flow Processor* (NFP), diseñados específicamente para procesamiento de flujos a alta velocidad y despliegues de redes basados en servidores [38].

Las *Agilio SmartNICs* están construidas sobre las familias *NFP-4000* y *NFP-6000*, procesadores organizados en una arquitectura de islas interconectadas mediante un bus interno de alta velocidad (*CPP*, *Command Push Pull*). Entre sus principales bloques internos se encuentran:

- Islas de procesamiento (*Flow Processing Core Islands*), cada una con múltiples *FPCs* (llamados tambien *microengines*) capaces de procesar paquetes en paralelo. Se cuenta 60 *FPC* de los cuales el SDK permite cargar programas hasta en 54.
- *MACs* para las interfaces de red.
- Motores *DMA*, *packet classifier*, *traffic manager* y *packet modifier*.

- Unidades de memoria.

La *SmartNIC* utilizada en este trabajo utiliza *NFP-4000*.

Los *FPCs* son multihilo, cada uno con 8 *hardware contexts*, memoria propia de instrucciones y datos, y acceso directo a buses internos. Esta organización permite paralelizar el procesamiento de paquetes en múltiples núcleos. [38] [39]

Los *FPCs* pueden ser programados usando lenguajes de alto nivel como P4 y *C*, con el *Software Development Kit* (SDK) provisto por Netronome. El SDK incluye:

- Compilador de P4,
- compilador que traduce la representación intermedia (*IR*) de P4 a C,
- compilador C (*NFCC*) para generar firmware ejecutable en los *FPCs*,
- *linker* que que enlaza el código compilado para generar el *firmware* de *NFP*,
- *loader* que carga los archivos *NFFW* (*firmware*) al *NFP*,
- entre otros.

El *NFP* permite describir el plano de datos mediante P4, usando la sintaxis estándar definida por el consorcio P4.org. El compilador P4 produce:

- el grafo de *parseo*,
- los flujos de procesamiento de *ingress* y *egress*,
- la IR que será posteriormente traducida a C.

El SDK también genera una API de tiempo de ejecución para manipular tablas de coincidencia (*match-action*) [39].

La ejecución del código generado se distribuye en múltiples *FPCs*, cada uno procesando paquetes de manera independiente.

Dentro del *Netronome* SDK, el término *Micro-C* se utiliza para referirse a la variante del lenguaje C soportada por el compilador del *NFP* (*NFCC*) y ejecutada directamente por los *FPCs* [32]. Es un entorno reducido y especializado, distinto del C estándar: omite características como recursión, *function pointers*, *floating point* y gran parte de la librería estándar, y añade extensiones para ubicar datos en regiones de memoria específicas (*IMEM*, *CTM*, *CLS*, *EMEM*), controlar alineamientos, usar registros de transferencia y manejar señales de sincronización. El lenguaje incluye *intrinsics* para acceder a memorias, colas y mecanismos internos del *NFP*, permitiendo implementar lógica paralela y de bajo nivel sin recurrir a ensamblador [31].

El uso conjunto de P4 y C se realiza mediante *sandboxing*: acciones en P4 se definen como `primitive_action`, que se traducen a llamadas a funciones C en archivos separados del SDK [38].

3.2. ELK

El término ELK hace referencia al conjunto de tres proyectos de código abierto desarrollados por *Elastic*: *Elasticsearch* [14], *Logstash* [27] y *Kibana* [24]. En conjunto, conforman una plataforma integral para la ingestión, almacenamiento, procesamiento y visualización de datos de *logs* provenientes de diferentes sistemas. Este ecosistema permite realizar análisis en tiempo real y obtener información útil a partir de los eventos registrados, facilitando la toma de decisiones basada en datos.

Los componentes del *stack* ELK cumplen funciones específicas y complementarias:

- ***Elasticsearch***: motor de búsqueda y análisis distribuido que permite indexar y consultar grandes volúmenes de datos de manera eficiente. Ofrece una interfaz *RESTful* y soporta operaciones de agregación que facilitan la obtención de métricas y estadísticas en tiempo real.

- ***Logstash***: canal de procesamiento de datos del lado del servidor. Recopila información de diversas fuentes, la transforma mediante filtros configurables y la reenvía hacia *Elasticsearch* u otros destinos. Su arquitectura basada en *pipelines* permite normalizar, enriquecer o filtrar los datos antes de su almacenamiento.

- ***Kibana***: interfaz de visualización que se ejecuta sobre *Elasticsearch*. Permite explorar y representar los datos indexados mediante gráficos, paneles interactivos y consultas dinámicas, favoreciendo la interpretación visual de los resultados.

El *stack* ELK se estructura como una canalización de datos que permite el flujo continuo de información desde las fuentes de origen hasta su visualización. De forma general, el proceso comienza con la recolección de *logs*, continúa con su procesamiento y normalización, y finaliza con su indexación y exploración visual, como se ilustra en la Figura 3.1.

Figura 3.1: Flujo general de datos en una arquitectura basada en ELK.

En esta arquitectura, cada componente desempeña un papel específico dentro del flujo de datos. En primer lugar, *Logstash* ingiere la información proveniente de diversas fuentes, la transforma mediante filtros y la envía al destino correspondiente. Luego, *Elasticsearch* recibe los datos, los indexa para permitir búsquedas eficientes y los analiza aplicando operaciones de agregación y consulta en tiempo real. Finalmente, *Kibana* proporciona la capa de visualización,

permitiendo representar los resultados del análisis mediante gráficos, paneles interactivos y consultas dinámicas.

3.3. DPDK y *Moongen*

Para algunos de los experimentos llevados a cabo en el presente trabajo se utilizó la herramienta *MoonGen* [15].

MoonGen es un generador de tráfico de alto rendimiento construido sobre DPDK [12], un conjunto de bibliotecas que permite a las aplicaciones de usuario enviar y recibir paquetes directamente desde la NIC (Network Interface Card) evitando el *kernel* y utilizando memoria de *huge pages* para maximizar el rendimiento. Sobre esta base, MoonGen utiliza *LuaJIT* [28] para construir y modificar paquetes en tiempo real, alcanzando tasas de transmisión de varios millones de paquetes por segundo por núcleo.

Además de transmitir, MoonGen puede recibir y clasificar paquetes, lo que habilita pruebas avanzadas, mediciones de pérdida y la emulación de patrones complejos de tráfico.

En este proyecto, MoonGen se empleó para enviar trazas (archivos `.pcap`) a tasas de paquetes por segundo que no podían ser alcanzadas por herramientas como *tcpreplay*.

3.4. Conclusiones

En este capítulo se describieron la plataforma experimental y las tecnologías empleadas para llevar a cabo el proyecto. Se analizaron las características de la *SmartNIC Netronome Agilio CX*, su arquitectura interna y las herramientas que proporciona su SDK para el desarrollo en P4 y *Micro-C*. Asimismo, se introdujeron las tecnologías que conforman la capa de visualización (stack ELK) y las herramientas utilizadas en la etapa de evaluación experimental (MoonGen y DPDK).

En el capítulo siguiente se presenta la arquitectura propuesta y se detalla su implementación concreta.

Capítulo 4

Arquitectura e implementación

Este capítulo constituye el núcleo del presente trabajo y describe en detalle el diseño, desarrollo e implementación de la plataforma de monitoreo basada en INT desarrollada sobre una *SmartNIC Netronome Agilio CX 2×10 GbE* siguiendo el artículo de referencia [19].

El objetivo de este capítulo es presentar el proceso completo de construcción del sistema, desde las decisiones arquitectónicas iniciales hasta las pruebas de funcionamiento y validación de los componentes desarrollados. A lo largo del capítulo se expone cómo se diseñó e implementó el INT *Sink*, encargado de recolectar y procesar la información de telemetría generada en la red, y cómo se integró con un sistema de almacenamiento y visualización basado en el *stack* ELK, que permite analizar las métricas recolectadas de forma flexible y en tiempo casi real.

Además, se presenta una prueba de concepto de un nodo intermedio (INT *transit hop*), con el fin de evaluar la viabilidad de instrumentar la totalidad del dominio INT sobre *hardware* programable.

La estructura del capítulo se organiza de la siguiente manera:

- En la Sección 3.1 se introduce la arquitectura general del sistema desarrollado, describiendo los planos que lo componen, sus objetivos y las principales decisiones de diseño adoptadas.
- En la Sección 3.2 se detalla el diseño e implementación del INT *Sink*, abordando tanto el desarrollo del *pipeline* P4 como de las rutinas de procesamiento en *Micro-C*, así como los mecanismos de comunicación entre la *SmartNIC* y el *host* de monitoreo.
- La Sección 3.3 presenta el sistema de almacenamiento y visualización, explicando el flujo de datos hacia el *stack* ELK y las visualizaciones implementadas.

- Finalmente, en la Sección 3.4 se describe la prueba de concepto del nodo intermedio, junto con los resultados obtenidos.

4.1. Arquitectura general de la plataforma desarrollada

Esta sección describe en detalle la arquitectura de la plataforma de monitoreo basada en INT desarrollada. El sistema tiene como finalidad obtener métricas de desempeño e información detallada sobre el estado y comportamiento de la red directamente desde el plano de datos, sin requerir sondas o tráfico de medición adicional, y procesarlas en tiempo casi real mediante una infraestructura de análisis y visualización flexible.

La arquitectura propuesta combina técnicas de programación en el plano de datos con tecnologías modernas de almacenamiento y visualización de métricas como lo son las que componen el *stack* ELK. Esta combinación permite recolectar información granular de los flujos de red, procesarla en el propio dispositivo de red y ofrecer visualizaciones de alto nivel orientadas tanto al diagnóstico como a la investigación de patrones de tráfico.

4.1.1. Motivación y objetivos del diseño

El enfoque tradicional de monitoreo en redes, basado en herramientas como SNMP o RMON, se caracteriza por mediciones de baja frecuencia y una visión parcial del comportamiento del tráfico. Por el contrario, INT ofrece la posibilidad de embeber instrucciones de telemetría dentro de los paquetes de datos, de modo que cada elemento de red que los atraviesa puede añadir información sobre su propio estado (latencia, ocupación de cola, identificación de interfaz, etc.). Este enfoque no solo posibilita una monitoreo detallada y granular del desempeño de la red, sino que además proporciona mediciones precisas y en tiempo real desde los propios paquetes de usuario, mejorando las prácticas tradicionales basadas en SNMP o RMON. La información recolectada mediante INT puede también emplearse para optimizar el control de congestión y detectar eventos complejos derivados del análisis acumulativo del recorrido de los paquetes a través de los distintos elementos de red [18].

En el contexto de INT, el lenguaje P4 aprovecha los avances recientes en la capacidad de cómputo de los dispositivos de red -como *switches*, *routers* o tarjetas de red (*NICs*)- para mejorar el monitoreo. Esto se logra al insertar campos adicionales en los paquetes que permiten a cada *switch* agregar información. Sin embargo, procesar todos estos campos de telemetría acumulados a lo largo de la ruta de un paquete y entregar la información, representa un desafío considerable, dado el volumen y granularidad de los datos generados.

Con la necesidad de anchos de banda cada vez más altos, los servidores dependen cada vez más de *SmartNICs* para delegar el procesamiento de los paquetes de red. Las *SmartNICs* permiten acelerar las operaciones de red al

ejecutar tareas directamente en el plano de datos, liberando recursos de la *CPU* principal y mejorando el rendimiento general del sistema [18].

Uno de los tantos desafíos que se presentan radica en el extremo receptor: el INT *Sink* debe procesar un volumen muy alto de paquetes con información de telemetría, extraer los metadatos, agregarlos por flujo y exportarlos a un sistema de almacenamiento, intentando que el paquete original siga su recorrido sin verse afectado, es decir, que el INT *Sink* funcione como un "*bump in the wire*". Las soluciones en las que la *CPU* del host se encarga del procesamiento de los paquetes INT usualmente tienen problemas para poder realizar el procesamiento a tasa de linea (*line rate*).

Además, realizar dicho procesamiento en *CPU* consume capacidad de cómputo que podría destinarse a otras tareas del sistema, lo cual es altamente deseable en ambientes de *Cloud Computing*.

Por esos motivos el presente proyecto adopta un enfoque que hace uso de *SmartNICs* para el procesamiento en el *Sink*, tomando como referencia la plataforma propuesta en [19] y, además, extendiendola en dos sentidos:

1. Incorporando un *pipeline* de visualización en tiempo real basado en ELK, en lugar del almacenamiento en *Redis* propuesto en el trabajo original.

2. Añadiendo campos personalizados (`request_id` e `is_response`) para habilitar correlación de flujos de solicitud y respuesta, orientada al análisis de comportamiento de aplicaciones.

En conjunto, estos componentes conforman la arquitectura completa del nodo final, desde la captura y procesamiento de los paquetes INT hasta la presentación y visualización de métricas y eventos.

A su vez, a través de la prueba de concepto de los nodos de tránsito INT (los que insertan metadatos en los paquetes) se exploran los demás componentes necesarios para una arquitectura INT completa.

4.1.2. Descripción de la topología

La infraestructura desarrollada para este proyecto se concibe como una red en la que existen nodos instrumentados con capacidades de INT. En particular, se implementa el modo *INT-MD* (2.4.6) utilizando el esquema de "INT over TCP/UDP" (2.4.5).
En este modo, los paquetes transportan los campos de telemetría directamente dentro de su flujo de datos: en el caso de TCP, se inserta un nuevo encabezado UDP e INT entre las capas *IP* y TCP; mientras que para UDP se reutiliza el encabezado original con un puerto de destino reservado para indicar la presencia de información INT.

Dentro de esta topología, los nodos que implementan INT pueden clasificarse, de acuerdo a su rol, como *Source*, *Transit Hop* o *Sink*. El INT *Source* es el punto donde se insertan en los paquetes las instrucciones de telemetría que determinan qué métricas deben recolectarse a lo largo de la ruta. Los *Transit Hops* son los nodos intermedios encargados de ejecutar esas instrucciones y agregar

sus propios metadatos. Finalmente, el INT *Sink* es el nodo donde se concentran los paquetes instrumentados y se extrae toda la información recolectada durante el trayecto. Esto se explica con más detalle en la sección 2.4.

En el contexto de este trabajo, se implementó de forma aislada el nodo INT *Sink*. Para realizar las pruebas y experimentos, se utilizaron trazas de tráfico reales a las que se les inyectó información de telemetría, emulando el recorrido de los paquetes a través de múltiples nodos INT. Esto permitió analizar el comportamiento del *Sink* en condiciones equivalentes a las de una red con varios saltos instrumentados, sin requerir la implementación física de todos los nodos.

La topología general de la infraestructura del *Sink* puede resumirse de la siguiente forma:

- Un generador de tráfico, encargado de reproducir trazas con encabezados INT previamente inyectados.
- El nodo INT *Sink*, implementado sobre una *SmartNIC Netronome Agilio CX 2×10 GbE*, que procesa los paquetes, extrae los metadatos y los exporta al sistema de análisis.
- Un host asociado a la *SmartNIC*, que recibe los datos de telemetría y los reenvía al *pipeline* de almacenamiento y visualización basado en ELK.

Además, de manera complementaria, se desarrolló una prueba de concepto de un nodo intermedio. Dicha implementación no interactúa directamente con el *Sink* desarrollado.

4.2. Diseño e implementación del INT *Sink*

El INT *Sink* constituye el componente central del sistema desarrollado, encargado de procesar los paquetes instrumentados que llegan desde la red, extraer los metadatos de telemetría embebidos y agregarlos por flujo antes de exportarlos al host para su almacenamiento y visualización.

El diseño adopta un enfoque híbrido que combina el procesamiento basado en P4 con rutinas programadas en *Micro-C* en la *SmartNIC*. Esta separación de responsabilidades busca delegar a cada entorno las tareas para las que resulta más eficiente.

En esta arquitectura, el P4 se encarga del *parseo* de encabezados y la clasificación inicial de los paquetes, identificando aquellos que contienen información INT. Una vez identificado un paquete como INT, los metadatos contenidos en el bloque *INT-MD* se extraen y son enviados a las rutinas *Micro-C*, donde se realiza la agregación por flujos. De esta forma, el P4 actúa como punto de entrada y extracción de campos, mientras que *Micro-C* implementa la lógica compleja de actualización, agregación y exportación de la información recolectada.

La figura 4.1 muestra un diagrama con el flujo lógico dentro del *Sink*. Los paquetes entran, son procesados y salen. Durante ese procesamiento, luego de haber *parseado* y extraído los metadatos INT, el programa P4 hace una llamada

a *Micro-C* (representada por la flecha verde etiquetada como `save_in_hash()`) para que estos metadatos pasen por el proceso de agregación y guardado en la tabla de *hash*. Eventualmente, la rutina de *Micro-C* mueve las entradas desde la tabla de *hash* hacia los *ring buffers*, ya sea de manera sincrónica (por *evicción*, representado con la flecha amarilla etiqueatada como `evict_to_rg()`) o asíncrona (por *age-out*, representado por la flecha amarilla etiquetada como `age_out_entry()`). El *host* lee de estos *ring buffers* (flecha azul grisaceo, etiquetada como `read_rings()`).
Todas estas estructuras de datos y operaciones se explican a continuación.

Estructuras de datos. El procesamiento se organiza en torno a una tabla *hash* llamada *FlowCache* (o H_P), residente en la memoria externa de la *SmartNIC*. Esta estructura contiene un conjunto de filas indexadas por el *hash* de la 5-tupla de cada flujo (direcciones *IP* de origen y destino, puertos de transporte y protocolo), y en cada fila se almacenan múltiples *buckets* que agrupan la información asociada a distintos flujos. Cada *bucket* incluye:

- La clave del flujo (*flow key*).
- La marca de tiempo del ingreso primer paquete del flujo.
- La marca de tiempo de la última actualización.
- Las métricas recolectadas por nodo, separadas en dos casos:
  - Último valor reportado.
  - Promedio de valores reportados.
- Un contador de paquetes procesados.
- Un identificador de la *request* junto con una *flag* que indica si se trata de una *request* o una *response*. (Los campos `request_id` `is_response` mencionados en secciones anteriores.)

La tabla de *hash* H_P tiene 2^{18} filas, cada una de las cuales tiene 12 *buckets*.

Operaciones sobre H_P. Cada paquete entrante puede producir dos tipos de operaciones:

1. **Actualización**: si el flujo ya existe o hay un *bucket* libre en la fila, se actualizan los contadores, las métricas y el *timestamp* de actualización, y el *timestamp* del primer paquete en caso que sea el primer paquete del *bucket*.
2. **Evicción**: si todas las entradas de la fila están ocupadas y el flujo es nuevo, se selecciona una entrada a ser reemplazada, siguiendo una política definida más adelante. Antes de hacerlo, su contenido es exportado al *host* mediante un *ring buffer*.

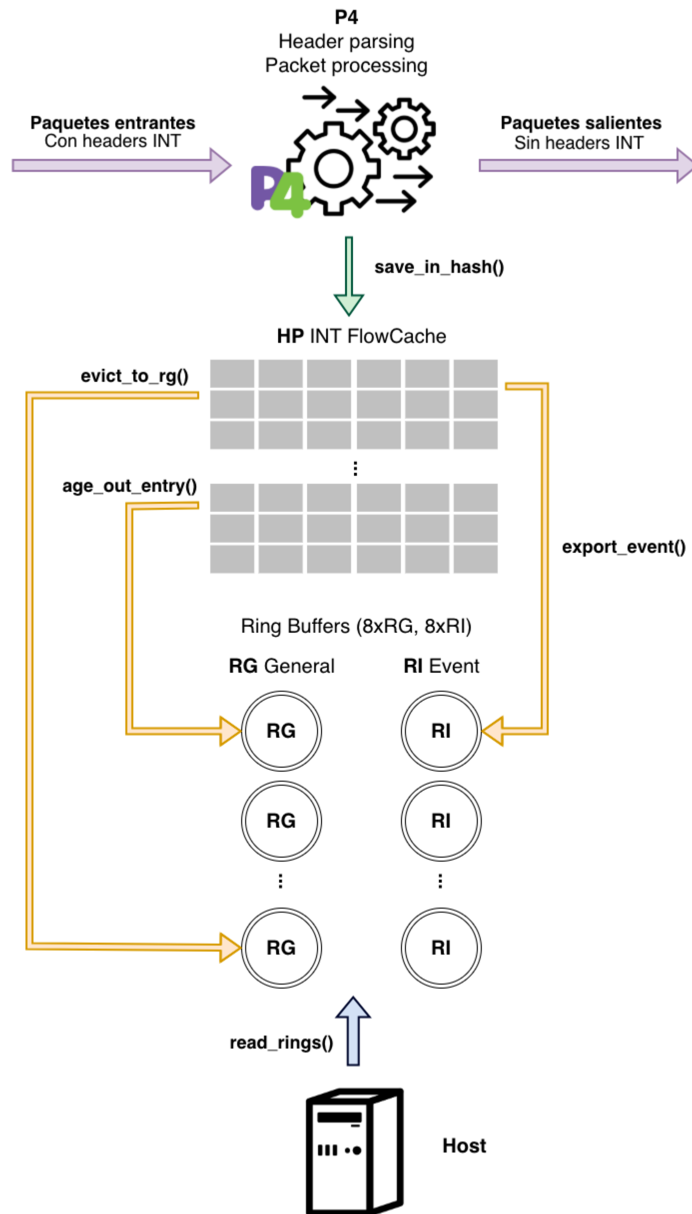


Figura 4.1: Diagrama de flujo del INT *Sink*.

***Ring buffers* y comunicación con el *host*.** El intercambio de información entre la *SmartNIC* y el *host* se realiza mediante *ring buffers*. Un *ring buffer* es una cola circular que reside en memoria global de la *SmartNIC* y que permite exportar información hacia el *host* sin bloquear el procesamiento de paquetes entrantes. Existen dos tipos de *ring buffers*:

- ***General ring buffers* (RG):** Utilizados para exportar los registros de flujos que fueron desplazados desde la tabla H_P, ya sea porque ocurrió una colisión y no quedaba espacio disponible en la fila correspondiente, o porque un proceso asíncrono los movió al detectarse que su última actualización ocurrió hace más tiempo que un umbral establecido (*age-out*).
- ***Event ring buffers* (RI):** Empleados para exportar eventos de telemetría detectados en tiempo real, como cambios abruptos o superación de umbrales en las métricas recolectadas.

Cada *ring buffer* puede almacenar hasta 2^{17} entradas y es leído continuamente por hilos en el *host*. En total se cuenta con 8 RGs y 8 RIs. Estos números están basados en los reportados en el articulo de referencia [19], el cual explica que con 2^{16} se garantiza que no se desborden las colas y provoque perdida de paquetes.

Procesamiento de eventos. Las rutinas en *Micro-C* realizan además tareas de detección de eventos y agregación temporal. Cuando una métrica excede un umbral configurado o cambia significativamente respecto al valor anterior, se genera un evento (de tipo *Threshold crossing* o *Change event*). Estos eventos son exportados a los RIs y posteriormente leídos por el *host*.

Procesamiento en el *host*. El *host* asociado a la *SmartNIC* se encarga de leer de forma concurrente las colas RG y RI, y transferir los datos hacia el sistema de análisis. En la sección 4.3.1 se describe en detalle el *pipeline* que transfiere los datos leídos por el *host* hacia el módulo de almacenamiento y visualización.

Mientras que en la arquitectura de referencia original los datos son almacenados en una base en memoria (*Redis* [40]), en este proyecto se extiende la plataforma incorporando un *pipeline* basado en ELK que permite almacenar, consultar y visualizar los datos de telemetría con mayor flexibilidad y capacidad analítica.

Habiendo presentado en alto nivel la estructura general del *Sink*, en las siguientes secciones se profundiza en los detalles concretos de la implementación.

En esta implementación se consideran cuatro metadatos por nodo (*node id*, *hop latency*, *queue occupancy* y *egress interface TX utilization*), que corresponden a los campos utilizados en la arquitectura objetivo. No obstante, el diseño del *Sink* es extensible y permite incorporar otros metadatos definidos por la especificación INT sin modificar su estructura general.

4.2.1. Procesamiento en P4

El programa P4 implementado en la *SmartNIC* cumple la función de realizar el *parseo* y extracción de los campos INT embebidos en los paquetes, así como la generación de la información necesaria para que las rutinas *Micro-C* realicen posteriormente el procesamiento y la agregación de flujos.

Su diseño se basa en la especificación INT-MD (con INT sobre TCP/UDP) definida por la versión 2.1 del estándar [36], y sigue una estructura modular compuesta por los bloques clásicos del modelo `v1model` [11]: *parser*, *ingress*, *egress*, *checksum computation* y *deparser*. En la Figura 4.2 se puede ver la arquitectura del `v1model`. La Figura 4.3, por otro lado, presenta una visión de alto nivel del procesamiento en P4 en el INT *Sink*, cuyos detalles se abordan a lo largo de esta sección.

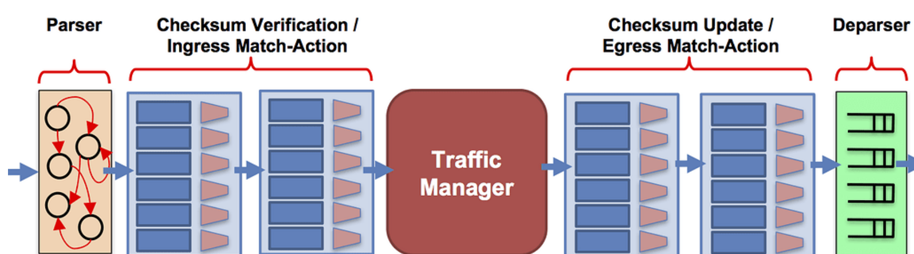


Figura 4.2: `v1model` [11].

Estructura general. El programa incluye los *headers* estándar de capa 2–4 (`ethernet`, `ipv4`, `udp`, `tcp`), junto con los *headers* específicos de INT: el `intl4_shim_t`, el `int_header_t` y una pila de `stack_element_t` para almacenar los bloques de metadatos de cada nodo que participó en el camino INT. El *parser* está diseñado para reconocer secuencialmente estos *headers*, determinar el número de nodos instrumentados y extraer los datos correspondientes a cada uno.

***Parser* y extracción de campos.** El bloque `MyParser` inicia la lectura del paquete extrayendo el *header ethernet* de la capa de enlace. Si el campo `ethernet type` es IPv4 transiciona al estado que *parsea* IPv4. Tras extraer el *header* de IPv4, si el protcolo es UDP se transiciona al estado que *parsea* UDP.
Si el datagrama UDP tiene puerto de destino `INT_TBD` (al no estar definido aún en la especificación, decidimos usar el puerto 5000) reservado para tráfico INT, el *parser* extrae el *header* `INT-L4 Shim` y el *header* principal INT *Header*. En esta etapa se determina el número de nodos que participaron en el dominio INT (`nodes_present`).

El *parser* continúa extrayendo los datos de cada nodo de manera iterativa. Para ello define un conjunto de estados `parse_nodeX_entry`, `parse_nodeX_loop` y `parse_nodeX_after` que permiten recorrer dinámicamente los bloques

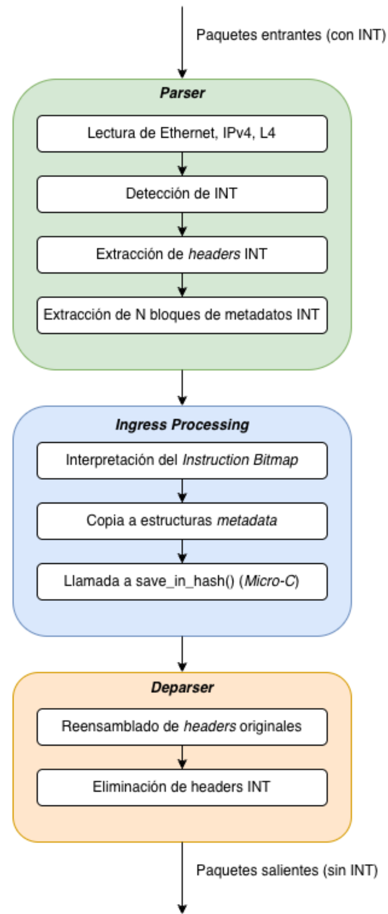


Figura 4.3: Diagrama de alto nivel del procesamiento en P4.

de metadatos de hasta cinco nodos (`MAX_INT_NODES = 5`). Cada palabra de 4 bytes (32 bits) leída se almacena temporalmente en los arreglos `hdr.nodeX_raw_data`, preservando el orden de los campos de telemetría.

Finalmente, el *parser* identifica el protocolo de capa superior original (TCP o UDP) y, en caso de encontrarse un campo de aplicación, extrae un *header* adicional denominado `app_metadata_t`, que incluye el *id* de la *request*, así como una *flag* que indica si es una *request* o una *response*.

En la figura 4.4 se muestra una máquina de estados con las transiciones entre los distintos estados del *parser*.

Bloque de *Ingress Processing*. El bloque `MyIngress` constituye la lógica principal del procesamiento en P4. Su propósito es interpretar las instrucciones

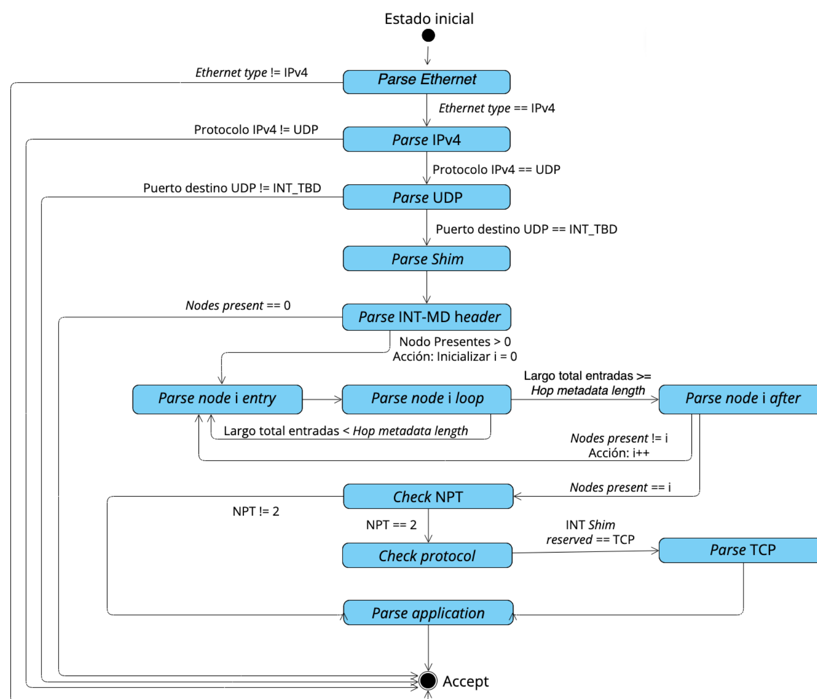


Figura 4.4: Máquina de estados: *parser* del *Sink*.

del *header* INT *Header*, extraer los campos solicitados y almacenarlos en la estructura interna *metadata*, que será utilizada posteriormente por el firmware *Micro-C*.

Para ello, el campo `instruction` del *header* INT se interpreta como un *bitmap* en el que cada bit activa la lectura de una métrica particular: identificador de nodo, interfaces de nivel 1 y 2, latencia por salto, ocupación de cola, *timestamps* de ingreso y egreso, y *buffer occupancy*, entre otras. En función de los bits activados, se ejecutan acciones específicas (`populate_node_id_metadata()`, `populate_hop_latency_metadata()`, `populate_queue_occupancy_metadata()`, etc.) que copian los valores desde los campos crudos (`nodeX_raw_data`) hacia las estructuras tipadas de `nodeX_metadata`.

El mecanismo de `pop_from_stacks()` se utiliza para ir desplazando la pila de datos conforme se procesan los campos, garantizando que las lecturas se realicen en el orden correcto dentro del bloque de metadatos.

Una vez completada la extracción de todos los campos requeridos, el programa invoca la función externa `save_in_hash()`, implementada en *Micro-C*. Esta llamada constituye el punto de enlace entre el plano P4 y *Micro-C*, y provoca el envío de la información recolectada hacia la función encargada de almacenar los datos en la tabla H_P. De esta forma, el código en P4 actúa como un módulo de extracción y preprocesamiento dentro del INT *Sink*. No realiza cálculos,

agregaciones ni comparaciones: su tarea consiste únicamente en identificar los paquetes relevantes, decodificar los *headers* INT y estructurar los metadatos para ser procesados por las rutinas de *Micro-C*. Una vez que los datos son enviados a *Micro-C* mediante `save_in_hash()`, las funciones en *Micro-C* se encargan de insertarlos o actualizarlos en la tabla H_P, y eventualmente exportarlos a los *ring buffers* en caso de evicción, detección de eventos o *age-out*.
Por último, se marca el puerto por el que va salir el paquete.

Deparser y recomposición del paquete. Finalmente, el bloque `MyDeparser` reensambla los paquetes antes de su reenvío, emitiendo nuevamente todos los *headers* en el orden original: `Ethernet`, `IPv4`, UDP/TCP y metadatos de aplicación.
Los *headers* INT no se emiten. Uno de los roles del *Sink* es remover estos *headers* así como el *stack*.

4.2.2. Procesamiento en Micro-C

La lógica implementada en *Micro-C* representa el corazón del *Sink*, aquí es donde se clasifica los flujos entrantes, se generan los *buckets* con la información agregada por cada paquete y se detectan eventos.

A continuación se presenta una secuencia de pasos que resume el funcionamiento general del algoritmo de procesamiento. En el Anexo A.1 se puede encontrar el pseudocódigo correspondiente.

Algoritmo. El procesamiento realizado por el *Sink* sigue, de manera resumida, las siguientes etapas:

1. **Cálculo del identificador del flujo.** A partir de los encabezados del paquete se construye la clave de hash correspondiente al flujo y se obtiene la posición en la estructura *FlowCache* (H_P) donde se almacenará o actualizará su estado.

2. **Selección o reemplazo de la entrada del flujo.** Se inspeccionan los *buckets* asociados a la posición calculada para localizar una entrada vacía, una entrada con la misma clave o una entrada envejecida que pueda ser reemplazada. Si no se encuentra una opción válida, se selecciona la entrada más antigua como víctima.

3. **Exportación de entradas reemplazadas.** Cuando corresponde efectuar un reemplazo, el contenido de la entrada descartada se copia en un *ring buffer* utilizado para la exportación de datos hacia el host, y luego se limpia la entrada en H_P.

4. **Inicialización o actualización del estado del flujo.** En función de si el paquete es el primero observado para ese flujo o uno adicional, se almacenan los metadatos iniciales o se actualizan los contadores y valores agregados asociados.

5. **Procesamiento de las métricas INT por nodo.** Para cada nodo que aportó metadatos INT en el paquete, se actualizan los valores agregados (por ejemplo, promedios o último valor observado). Además, se comparan los valores actuales con los previos para detectar variaciones significativas o condiciones fuera de rango, registrando eventos cuando corresponda.

6. **Cálculo de métricas extremo a extremo.** Si el paquete contiene información suficiente, se calculan métricas globales del flujo, como la latencia extremo a extremo. Variaciones absolutas o relativas que superen umbrales predefinidos también generan eventos.

Mecanismo de detección de flujo (5-tupla) y clasificación. Para poder indexar un flujo entrante en H_P, se extrae la 5-tupla de los metadatos de P4. Para esto P4 brinda funciones especiales como `pif_plugin_hdr_get_ipv4()` para recuperar la dirección ipv4. Una vez se tiene los campos necesarios, se usa la función brindada por el SDK de *Netronome* `hash_me_crc32()`, cuyo resultado se le aplica una mascara para que el valor caiga en el rango de H_P, el valor final queda definido como:

```
hash_value = hash_me_crc32(FLOW_KEY) & (FLOWCACHE_ROWS - 1)
```

Para cuando se da el caso en que todos los *buckets* están llenos y se debe desplazar alguno hacia RG, se realiza una mascara nuevamente:

```
ring_index = hash_value & (NUM_RINGS - 1)
```

y de esta forma se reparten los flujos desplazados en los 8 RG.

Concurrencia. Debido a la naturaleza paralela con la que se trabaja y el uso de memoria global compartida, es de vital importancia hacer un control estricto de la concurrencia y los accesos a zonas compartidas.

Para conseguir esto se definió una estructura en memoria global que representa un semáforo binario por fila de H_P:

```
global_semaphores[FLOWCACHE_ROWS]
```

Para trabajar con este semáforo se utilizan las funciones `semaphore_up()` y `semaphore_down()`, que reciben una dirección de memoria y se bloquean o acceden, dichas funciones fueron tomadas de guías oficiales de *Netronome* [46]. Cuando un hilo quiere acceder a la fila que le corresponde simplemente debe ejecutar:

```
semaphore_down(&global_semaphores[hash_value])
```

El acceso a los *ring buffers* también debe ser controlado, por lo que se definió `ring_buffer_sem_X[NUM_RINGS]`, donde X puede ser I o G, representando RI o RG respectivamente.

Mecanismo de *Age-Out* para evicción de flujos. El *Sink* implementa dos mecanismos complementarios para la evicción de entradas del *flow cache*. El primero es la evicción sincrónica, que ocurre durante el procesamiento de un paquete cuando se detecta una colisión y no quedan *buckets* libres en la fila correspondiente. En ese caso, el sistema selecciona una entrada existente para su reemplazo y exporta sus metadatos al *ring buffer* antes de liberarla.

El segundo es el *age-out*, cuyo objetivo es eliminar de forma asíncrona aquellas entradas que han permanecido inactivas más allá de un umbral temporal predefinido. A diferencia de la evicción sincrónica, el *age-out* no depende de la llegada de nuevos paquetes: se ejecuta continuamente en *microengines* dedicados, que recorren particiones de H_P, comparan el timestamp del último paquete del flujo con el umbral (`AGE_THRESHOLD_NS`) y determinan qué entradas deben liberarse.

Cuando una entrada vence, sus metadatos se copian al *ring buffer* asociado y luego la entrada se limpia, quedando disponible para futuros flujos. Este procedimiento se realiza bajo semáforos que aseguran consistencia entre el procesamiento de paquetes y la rutina de limpieza.

Organización de hilos y *microengines*. Debido al alto uso de registros que cada hilo requiere, se utilizan solo 4 hilos de los 8 posibles. Cabe aclarar que este es el modo de operación por defecto en el compilador del SDK y también la misma cantidad usada en el articulo de referencia.

El programa principal encargado del procesamiento principal de los paquetes es cargado en 52 de los 54 *microengines* disponibles, quedando en los 2 restantes el programa encargado de mover entradas vencidas de H_p hacia los *ring buffers* (*age-out*). En el programa principal no hay diferenciación entre hilos o *microengines*, cada uno procesa un paquete de forma paralela. Mientras que en el *age-out* se divide el trabajo hecho en H_p en dos, donde un *microengine* procesa la primera mitad y el otro la segunda mitad, de esta forma se procesan más rápido las entradas y pierden menos paquetes.

Transformación de *ticks* a nanosegundos. Para que los *timestamps* de última actualización o ingreso del primer paquete de un flujo sean útiles a los operadores que visualicen los datos sería deseable que se encuentren en una unidad legible como lo puede ser `YYYY-MM-DD HH:MM:SS`.
Sin embargo, en la función que provee la para obtener el *timestamp* actual (`me_tsc_read()`), el mismo está expresado en *ticks*. Los *ticks* aumentan cada 16 ciclos del *clock* de los *microengines* de la *SmartNIC*, y se inicializan en 0 al cargar un nuevo *firmware*. De modo que, para poder pasarlo a unidades de tiempo del Sistema Internacional de Unidades (segundos, mili-segundos, etc.) lo primero que debemos conocer es la frecuencia del *clock* de los *microengines*.
Utilizando el comando `nfp-hwinfo` se puede obtener la frecuencia de los *microengines* expresada en MHz. En nuestro caso:

```
$ sudo /opt/netronome/bin/nfp-hwinfo me.speed
```

```
me.speed=633
```

En este caso, la frecuencia reportada corresponde a 633 MHz, lo que significa que el reloj de los *microengines* ejecuta 633×10^6 ciclos por segundo. Dado que cada *tick* equivale a 16 ciclos de reloj, la conversión entre *ticks* y nanosegundos se obtiene mediante:

$$\text{segundos_por_ciclo} = \frac{1}{633 \times 10^6}$$

$$\text{ns_por_ciclo} = \frac{10^9}{633 \times 10^6}$$

$$\text{ns_por_tick} = \frac{16 \times 10^9}{633 \times 10^6} = \frac{16 \times 1000}{633} \approx 25{,}28 \text{ ns/tick}$$

En el programa de *Micro-C* se cuenta con una función que se encarga de esa conversión y es usada para que en la tabla H_P se guarden *timestamps* en nanosegundos.

Por otro lado, los valores de tiempo expresados en nanosegundos son relativos al instante en que se cargó el *firmware* en la *SmartNIC*. Para poder asociar estos valores con una referencia temporal real (formato `YYYY-MM-DD HH:MM:SS`), durante el proceso de carga del *firmware* se exporta el tiempo local del *host*. Luego, en la etapa de preprocesamiento ejecutada en el *host* (previa al envío de los datos al *stack* ELK) se calcula el desplazamiento entre ese tiempo inicial exportado y los nanosegundos de los *timestamps* almacenados en H_P. De esta forma, se obtiene una *timestamp* absoluto que permite correlacionar los eventos de la *SmartNIC* con las mediciones y registros externos del sistema.

4.2.3. Comunicación con el host y exportación de datos

Como se mencionó anteriormente, la comunicación entre la *SmartNIC* y el *host* se realiza mediante *ring buffers* residentes en la memoria de la tarjeta. Sin embargo, leer directamente estos *rings* y escribir sus contenidos en el sistema de almacenamiento resultaría ineficiente y acoplaría fuertemente el ritmo de producción de la placa con el rendimiento de disco. Para evitar este problema, se introduce un componente intermedio en el *host*, denominado *Spooler*, cuya función es desacoplar ambos ritmos y ofrecer una interfaz estable hacia el *pipeline* de monitoreo basado en ELK.

Lectura de los *ring buffers* en el host. La aplicación `host_reader` es la encargada de interactuar con la *SmartNIC* y drenar los *ring buffers* exportados por el firmware. Al iniciarse, el programa abre el dispositivo *Netronome*, obtiene un manejador CPP y localiza los símbolos de tiempo de ejecución que describen tanto las áreas de datos de los *rings* (`_ring_buffer_G` y `_ring_buffer_I`) como sus metadatos (`_ring_G` y `_ring_I`). A partir de estas direcciones, se crean áreas

CPP que permiten mapear, desde el *host*, las estructuras de datos que mantienen la lista de entradas y los punteros de lectura y escritura de cada *ring*.

En el caso de los RG, el programa ejecuta un bucle principal que recorre todos los *rings* generales. Para cada uno, lee su estructura de metadatos, recuperando el *write pointer*, el *read pointer* y el indicador de llenado (`full`). Mientras existan entradas pendientes, se leen por lotes (`BATCH_SIZE`) las estructuras `bucket_entry` y se entregan al *Spooler* mediante la llamada `spooler_enqueue()`. Tras consumir un conjunto de entradas, el *host* actualiza los punteros de lectura y el indicador de llenado en la propia estructura de metadatos, de modo que el firmware pueda reutilizar el espacio en el *ring* sin ambigüedad.

Por otro lado, los RI se gestionan mediante hilos dedicados. La aplicación `host_reader` crea varios hilos de trabajo (`event_ring_worker`), parametrizados por la opción `-X` donde `X` $\in \{1, 2, 4, 8\}$, que se reparten los *rings* de eventos y los drenan en paralelo. Esta organización permite las siguientes ventajas: (1) reducir la latencia entre la detección de un evento en la *SmartNIC* y su persistencia en disco y (2) balancear la carga de trabajo necesario para cada RI entre los hilos y evitar perdida de paquetes. Todo esto sin modificar la lógica del *Sink*.

Modelo productor–consumidor y colas acotadas. El *Spooler* implementa un patrón clásico de productor–consumidor con cola acotada en memoria. En el caso de las métricas, existe una única cola global (`bucket_queue`) y un hilo consumidor; los productores son los hilos que leen los *ring buffers* generales y, por cada `bucket_entry` válido, invocan `spooler_enqueue()`. Esta función inserta la entrada en la cola protegida por un *mutex* y dos variables de condición (`not_empty` y `not_full`). Si la cola se encuentra llena, el productor se bloquea hasta que el hilo consumidor libere espacio.

Para los eventos, el diseño es ligeramente distinto: se crea una instancia de *Spooler* por cada *ring* de eventos. Cada instancia mantiene su propia cola (`event_queue`) y su propio hilo de escritura, lo que permite procesar en paralelo los eventos provenientes de distintos *rings* sin que un flujo de eventos especialmente activo bloquee al resto.

Segmentación de archivos y formato NDJSON. El hilo consumidor del *Spooler* no escribe en un único archivo creciente, sino que organiza los datos en segmentos consecutivos en disco. Cada segmento se crea inicialmente con extensión `.ndjson.open`, se le van agregando objetos y, una vez que se cumple alguna condición de rotación, se cierra y se renombra de forma atómica a `.ndjson`. Este esquema garantiza que *Filebeat* sólo procese archivos completamente escritos, evitando condiciones de carrera y registros parcialmente generados.

La rotación de segmentos está gobernada por tres límites:

- Un tamaño máximo en bytes (`SEG_MAX_BYTES`).
- Un número máximo de documentos por archivo (`SEG_MAX_DOCS`).
- Una antigüedad máxima del segmento (`SEG_MAX_AGE_MS`).

Mientras se escribe, cada entrada del *Spooler* se traduce en exactamente un objeto JSON por línea. En el caso de las métricas, cada objeto incluye, entre otros, los campos `@timestamp`, `host.name`, las claves del flujo, contadores de paquetes, marcas de tiempo crudas provenientes de la *SmartNIC* y las listas de métricas `int.latest` e `int.average`. Para los eventos, los objetos contienen el identificador del nodo, el valor medido, el *bitmap* de tipo de evento y la marca de tiempo asociada.

Además, la convención de nombres de los archivos incluye el *hostname*, un sello temporal y un número de secuencia; en el caso de los eventos, también se codifica el índice de *ring*. Esto facilita tanto la depuración manual como la trazabilidad de segmentos concretos dentro del *pipeline*. A modo ilustrativo, a continuación se muestran dos ejemplos de nombres generados por el sistema:

```
métricas: host=smartlab.ts=20251127T182615Z.seq=000001.ndjson
eventos:  host=smartlab.ring=8.ts=20251120T010322Z.seq=000053.ndjson
```

Gestión de marcas de tiempo y alineación temporal. Las marcas de tiempo generadas en la *SmartNIC* se encuentran en nanosegundos relativos al momento de carga del *firmware*. Como se mencionó anteriormente, el *Spooler* aplica un desplazamiento temporal que se obtiene durante la inicialización desde un archivo local que guarda la referencia de carga del programa. Al escribir cada documento NDJSON, se combinan los valores de `first_packet_ts` y `last_update_ts` con este desplazamiento y se convierten a formato `ISO8601`. De esta manera, las métricas y eventos procedentes de la *SmartNIC* se alinean con el tiempo del *host*, lo que permite correlacionarlos con otros registros.

Finalización ordenada y robustez. El ciclo de vida completo del *Spooler* se gestiona desde `host_reader`. Al inicio, se invocan `spooler_init()`, `spooler_start()`, `event_spooler_init()` y `event_spooler_start()`, lo que inicializa las colas y lanza los hilos de escritura. Durante la ejecución, una señal externa (por ejemplo, `SIGINT`) marca la variable global `stop`, indicando que el sistema debe detenerse. En respuesta, el programa despierta a todos los hilos que pudieran estar bloqueados en las colas, permite que los *workers* de eventos terminen de drenar sus *rings* y espera a que el *Spooler* consuma todas las entradas pendientes antes de cerrar y renombrar el último segmento NDJSON.

Este mecanismo de apagado ordenado evita la pérdida de datos en tránsito y garantiza que todos los archivos generados se encuentren en un estado consistente para ser procesados posteriormente por *Filebeat* y el resto del *stack* ELK.

4.2.4. Extensiones para casos de uso personalizados

La plataforma desarrollada está diseñada para ser extensible y admitir la incorporación de nuevos metadatos orientados a casos de uso específicos.

Como ejemplo de esta extensibilidad, se implementó una ampliación del conjunto de metadatos orientada a habilitar la correlación entre *requests* y *respon-*

ses. Para ello, se incorporan dos campos adicionales en la capa de aplicación: `request_id` e `is_response`. Estos campos permiten calcular el tiempo entre las solicitudes y sus respectivas respuestas, dado que ya se mantiene un registro de *timestamps*.
En este trabajo se asume que hay una request por flujo (el tiempo entre paquetes de una *request* y otra es mayor al *flow time gap* o cambia puerto el origen/destino).
Contemplar la existencia de una colección de *request ids* distintas dentro de un mismo flujo queda como una posible línea de trabajo a futuro.

Diseño e integración en la plataforma. Para su incorporación, se definió un nuevo *header* denominado `app_metadata_t`, agregado al final del conjunto de *headers* procesados por el programa P4. Este *header* contiene un campo de 24 bits denominado `request_metadata`, dentro del cual se codifican los valores de `request_id` e `is_response`. El formato actual asigna los 16 bits más significativos al identificador de solicitud (`request_id`) y un bit específico al indicador de respuesta (`is_response`), reservando los bits restantes para futuras extensiones y alineamiento de memoria.

El bloque `MyParser` se encarga de reconocer y extraer este *header* en la última etapa del *parseo* de los paquetes, luego de los *headers* de transporte (UDP/TCP) y de los bloques de metadatos INT. Una vez extraído, el campo `request_metadata` se mantiene disponible en el *metadata* interno para ser accedido durante el procesamiento en *Micro-C*.

El campo `request_metadata` se mantiene como un valor crudo de 24 bits tanto en la tabla H_P como en los *ring buffers*. La separación en sus componentes `request_id` e `is_response` no se realiza dentro del *firmware*, sino durante el procesamiento posterior en el *host*. Específicamente, el *pipeline* de *Logstash* se encarga de decodificar este campo, generando dos atributos diferenciados en los documentos almacenados en *Elasticsearch*. A partir de estos valores, el cálculo del tiempo transcurrido entre solicitudes y respuestas se lleva a cabo de manera diferida mediante consultas en *Elasticsearch*, lo que permite conservar la simplicidad del procesamiento en la *SmartNIC* y delegar las operaciones analíticas a una capa más alta de la arquitectura.

Notar que estos metadatos fueron agregados fuera de los *headers* INT. Esto se debe a varios motivos: (i) no forman parte de la especificación oficial, la cual es seguida por nuestra implementación; y (ii) semánticamente no correspondería incorporarlos, dado que los metadatos INT son agregados por los nodos del INT *domain* que atraviesa el tráfico, mientras que los metadatos de aplicación están destinados a ser manipulados únicamente por las aplicaciones de los extremos (y, en nuestro caso, por el nodo de monitoreo).

En ese sentido, extender la plataforma no requiere ajustarse estrictamente a la especificación INT. La arquitectura desplegada permite incorporar metadatos y mecanismos personalizados que atiendan necesidades específicas sin alterar el funcionamiento general del INT *Sink*.

4.3. Sistema de visualización y almacenamiento

El sistema de visualización y almacenamiento tiene como propósito central recolectar, procesar y hacer accesible la información generada por la placa de red programable. En la sección anterior se describió cómo el *Spooler* exporta tanto las métricas como los eventos INT que se encuentran en los *rings* de la *SmartNIC*, generando archivos en formato NDJSON en el *host*.

Sobre esta base, se construyó un pipeline de monitoreo apoyado en el conjunto de herramientas conocido como ELK *Stack*, complementado con *Filebeat* como agente de recolección. Este *pipeline* automatiza el flujo de información desde su captura hasta su representación visual, garantizando trazabilidad extremo a extremo, flexibilidad en las transformaciones y una estructura de datos coherente a lo largo de todo el proceso.

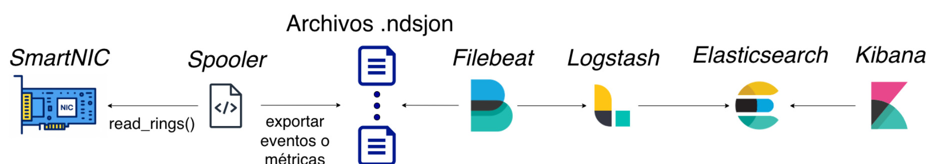


Figura 4.5: Arquitectura lógica del *pipeline* de monitoreo.

La figura 4.5 ilustra el flujo general de los datos: desde su generación en los *ring buffers* de la *SmartNIC* y su exportación en formato *NDJSON* por parte del *Spooler*, hasta su envío mediante *Filebeat* hacia *Logstash*, donde son procesados, transformados y finalmente almacenados en *Elasticsearch*. Una vez indexados, los documentos quedan disponibles para su exploración en *Kibana*, que permite analizarlos en tiempo (casi) real a través de *dashboards* y gráficos dinámicos.

Aunque en este trabajo todos los componentes del stack se despliegan en una única máquina, la arquitectura es intrínsecamente distribuida: *Filebeat*, *Logstash*, *Elasticsearch* y *Kibana* podrían ejecutarse en nodos distintos, permitiendo escalar de forma independiente la captura, el procesamiento y el almacenamiento. En conjunto, el *pipeline* conforma una cadena integral de monitoreo que va desde la captura a nivel de *hardware* hasta la representación visual a nivel de aplicación, proporcionando una visión completa y estructurada del comportamiento de la red.

4.3.1. Diseño del *pipeline* de monitoreo

El *pipeline* de monitoreo se compone de tres etapas principales: recolección, procesamiento e indexación/visualización. Cada una de ellas cumple una función específica dentro del flujo completo de datos exportados por el *Sink*.

Recolección: Filebeat

Filebeat actúa como agente de ingestión en el *host*. Su función es detectar los archivos NDJSON generados por el *Spooler*, leerlos de forma incremental y enviarlos hacia la etapa de procesamiento. De este modo, abstrae la gestión de archivos locales y asegura que los datos producidos por el *Sink* se transmitan de manera continua y confiable al resto del sistema.

Procesamiento: Logstash

Logstash recibe los documentos emitidos por *Filebeat* y aplica una normalización que convierte los registros generados por la *SmartNIC* en estructuras homogéneas y adecuadas para su análisis. En esta etapa se distinguen las muestras de flujo de los eventos INT, se reorganizan los campos relevantes y se incorporan etiquetas que permiten su clasificación posterior. El objetivo principal de esta fase es preparar los datos para que puedan ser indexados de forma eficiente y consultados mediante criterios semánticos.

Métricas INT (`int.metrics`). En el caso de las muestras de métricas, *Logstash* se encarga de transformar los valores compactados producidos por el *Spooler* en estructuras explícitas y fáciles de consultar. Algunos campos llegan empaquetados en enteros donde múltiples valores se codifican en distintos rangos de bits. Por ejemplo, el campo `queue_occupancy` combina en un único entero el identificador de la cola y su nivel de ocupación:

```
val = h["queue_occupancy"].to_i
h["queue"] = {
  "id"        => (val >> 24) & 0xFF,
  "occupancy" =>  val        & 0xFFFFFF
}
```

Esta decodificación convierte un valor opaco en una estructura `{id, occupancy}` adecuada para búsquedas y visualización. De forma análoga, las métricas por nodo contenidas en arreglos INT se reorganizan en columnas como `latest_hop_latency` o `average_egress_interface_tx`, lo que simplifica las consultas y facilita las agregaciones temporales en *Kibana*.

Indexación y exploración: *Elasticsearch* y *Kibana*

Una vez procesados, los documentos se almacenan en *Elasticsearch*, donde quedan disponibles para búsquedas, filtrado y agregaciones temporales. Sobre estos índices se construyen las visualizaciones en *Kibana*, que permiten explorar el comportamiento de los flujos y de los nodos INT en distintos niveles de detalle, utilizando gráficos dinámicos y *dashboards* temáticos.

4.3.2. ¿Por qué ELK?

Durante la etapa de diseño del sistema de monitoreo se evaluaron distintas alternativas para el almacenamiento y la visualización de los datos exportados por la *SmartNIC*. La arquitectura de referencia original empleaba una base en memoria *Redis* [40], adecuada para escenarios sencillos de monitoreo en tiempo real, pero con limitaciones a la hora de realizar consultas históricas o análisis complejos sobre grandes volúmenes de información.

La adopción del *stack* ELK resultó más adecuada para los objetivos de este trabajo por varias razones:

- ***Pipeline* integrado de ingestión y transformación.** *Filebeat* y *Logstash* permiten construir un *pipeline* explícito de ingestión y procesamiento, donde es posible decodificar campos, unificar formatos, enriquecer registros y descartar información redundante antes de su almacenamiento.

- **Almacenamiento y búsqueda flexibles.** *Elasticsearch* proporciona capacidades de búsqueda estructurada, agregaciones temporales y filtrado por múltiples campos, lo que resulta especialmente útil para analizar métricas INT a lo largo del tiempo y correlacionarlas con eventos.

- **Visualización nativa.** *Kibana* ofrece una integración directa con los índices, permitiendo construir *dashboards* interactivos sin programación adicional. Esto facilita la creación de vistas específicas para operadores de red, con filtros por nodo, flujo, tipo de evento o rango temporal.

- **Capacidad de despliegue distribuido.** Aunque en este trabajo todos los componentes se han desplegado localmente en una única máquina, el *stack* ELK está diseñado para entornos distribuidos, en los que *Filebeat*, *Logstash* y *Elasticsearch* pueden escalar de forma independiente según el volumen de datos y los requisitos de disponibilidad.

En síntesis, la elección de ELK permite ir más allá de la simple visualización de métricas en tiempo real: habilita la construcción de un sistema de observabilidad donde los datos INT pueden almacenarse históricamente, consultarse con flexibilidad y representarse mediante paneles adaptados a diferentes necesidades de análisis.

4.4. Prueba de concepto de nodo intermedio INT

Uno de los objetivos planteados en este trabajo fue implementar una prueba de concepto de un nodo intermedio INT, con el propósito de validar la viabilidad de desplegar un sistema de monitoreo basado en INT a lo largo de varios saltos de red.

Los nodos intermedios se ubican entre el nodo origen y el *Sink*, y son los responsables de inyectar en cada paquete las métricas locales que describen el estado de la red en el punto de medición: identificador del nodo, latencia de

salto, ocupación de cola, interfaces de entrada y salida, entre otros metadatos pedidos por el origen. De este modo, cada paquete va acumulando información a medida que atraviesa el dominio INT, lo que permite reconstruir a posteriori una visión detallada del camino recorrido y del estado de los recursos de red asociados.

La prueba de concepto desarrollada busca reproducir esta lógica de funcionamiento en un nodo intermedio genérico, de forma que el comportamiento sea coherente con la especificación INT-MD.

4.4.1. Funcionalidades de un nodo intermedio INT

Desde el punto de vista funcional, un nodo intermedio INT debe cumplir, al menos, con las siguientes tareas:

- **Interpretación del encabezado INT.** Leer el encabezado INT, validar su consistencia básica (por ejemplo, cantidad de saltos restantes y longitud de metadatos) y detectar qué campos fueron solicitados por el nodo origen a través del *instruction bitmap*.
- **Inserción de metadatos locales.** Extraer del plano de datos las métricas asociadas al nodo (identificador de dispositivo, timestamps de ingreso y egreso, latencia de salto, ocupación de cola, identificadores de interfaz, etc.) e insertarlas en la pila de metadatos INT respetando el formato y el orden definidos por la especificación.
- **Actualización de *flags* y contadores.** Actualizar los campos de control del encabezado INT, tales como el contador de saltos restantes, las banderas asociadas a condiciones especiales (*End of Stack*, *Mismatch*, etc.) y, en caso necesario, otros indicadores de estado.
- **Reensamblado y reenvío del paquete.** Ajustar los campos de longitud de los encabezados de red afectados (por ejemplo, `ipv4.total_len` y `udp.length`) para reflejar el nuevo tamaño del paquete y reenviarlo hacia el siguiente salto o hacia el *Sink*.

Estas funcionalidades permiten que el nodo intermedio se integre de forma transparente en el plano de datos: para el resto de la red, el paquete continúa su camino normal, pero ahora enriquecido con información de telemetría generada en cada salto.

4.4.2. Limitaciones en la *SmartNIC Netronome*

Durante el proceso de desarrollo se exploró la posibilidad de implementar un nodo intermedio INT directamente sobre la *SmartNIC Netronome* utilizada en el resto de la plataforma. Esta evaluación permitió determinar el alcance real del modelo de programación disponible para este dispositivo y su compatibilidad con los requisitos definidos por la especificación INT.

En este análisis se observó que el SDK y las abstracciones expuestas al programa P4 no permiten acceder a ciertos encabezados intrínsecos clave, como `intrinsic_metadata` o `queueing_metadata`. Estos encabezados son los que, en entornos compatibles, proporcionan información interna esencial para un nodo intermedio, tales como:

- Timestamps globales de ingreso y egreso del paquete.
- Latencia de cola o tiempo transcurrido entre encolado y desencolado.
- Profundidad de la cola en paquetes o bytes.
- Identificadores de cola y otras métricas internas del *hardware*.

La ausencia de estos metadatos implica que no es posible calcular de forma directa métricas fundamentales para un nodo intermedio INT, como la latencia por salto o la ocupación de cola. Como resultado, se concluye que esta *SmartNIC* no ofrece las capacidades necesarias para implementar un nodo intermedio conforme a la especificación.

A partir de esta evidencia, se decidió realizar la prueba de concepto en un entorno software que sí expone los metadatos requeridos de manera estandarizada, permitiendo evaluar el comportamiento funcional de un nodo intermedio INT sin las limitaciones presentes en la plataforma *hardware*.

4.4.3. Implementación de la prueba de concepto en BMv2

Para validar la lógica de un nodo intermedio completo se implementó la prueba de concepto en BMv2, el *software switch* de referencia para P4. Este entorno permite acceder a los encabezados intrínsecos definidos en el modelo de arquitectura, tales como `intrinsic_metadata` y `queueing_metadata`, y proporciona campos específicos para medir timestamps, profundidades de cola y otros parámetros internos del plano de datos.

Lógica básica de inserción de metadatos

El comportamiento del nodo intermedio en BMv2 sigue la semántica definida por la especificación INT-MD. En primer lugar, el nodo valida que el paquete tenga presupuesto de saltos (`remaining_hop_cnt >0`); de no ser así, activa la bandera `E` (*End of Stack*) y omite la modificación del encabezado INT. Asimismo, verifica que la longitud de metadatos por salto coincida con el valor de diseño (`hop_metadata_len = 4`); si difiere, marca `M` (*Mismatch*) y aborta la inserción.

La inserción de campos se rige por el `instruction bitmap` del encabezado INT. En función de los bits activados, el nodo:

- copia el identificador del dispositivo en el campo `node_id`,
- inserta timestamps de ingreso y egreso,

- calcula y escribe la latencia de salto,
- y actualiza métricas asociadas a la cola de salida y a las interfaces.

Tras agregar los metadatos requeridos, el nodo decrementa `remaining_hop_cnt`, incrementa el atributo `len` del *shim header* INT y ajusta los campos `ipv4.total_len` y `udp.length` para reflejar el nuevo tamaño del paquete, antes de reenviarlo hacia el siguiente salto del camino.

Recolección de métricas en BMv2

Para cada metadato solicitado por el nodo INT origen se definió una forma de obtenerlo en el entorno BMv2. A continuación se describe cómo se obtiene cada uno:

- **Node ID** (`node_id`): valor administrativo único por dispositivo INT, fijado por política (como constante en una acción, parámetro de tabla o registro).
- **Timestamp de ingreso** (`ingress_timestamp`): `intrinsic_metadata.ingress_global_timestamp` (µs).
- **Timestamp de egreso** (`egress_timestamp`): `intrinsic_metadata.egress_global_timestamp` (µs).
- **Latencia de salto** (`hop_latency`): preferentemente `queueing_metadata.deq_timedelta` (µs), que mide el tiempo que el paquete permaneció en la cola de salida. Alternativamente, como aproximación del tiempo total en el *pipeline* más la cola:

  $$\texttt{hop_latency} \approx \texttt{egress_global_timestamp} - \texttt{ingress_global_timestamp}$$

- **Ocupación de cola** (`queue_occupancy`) e **ID de cola** (`queue_id`):
  - `queue_occupancy`: `queueing_metadata.deq_qdepth` (en paquetes) al momento de desencolar. Alternativamente, `enq_qdepth` mide la condición al encolado.
  - `queue_id`: `queueing_metadata.qid` si el *hardware* expone múltiples colas; de lo contrario, puede fijarse por política local (por ejemplo, 0).
- **Interfaces**:
  - `ingress_interface_id`: `standard_metadata.ingress_port` (puerto físico de entrada).
  - `egress_interface_id`: en el bloque de *ingress*, `standard_metadata.egress_spec`; en *egress*, `standard_metadata.egress_port`.

- **Utilización del enlace de egreso** (`egress_interface_tx_utilization`): derivada de `standard_metadata.deq_timedelta` y de la longitud de los paquetes, lo que permite estimar la ocupación relativa del enlace en la ventana de tiempo observada.

La implementación en BMv2 permite, de esta forma, ejercitar todo el ciclo de vida de los metadatos INT en un nodo intermedio, verificando la correcta interacción entre el *instruction bitmap*, los encabezados intrínsecos y las estructuras de datos del plano de datos, aun cuando la plataforma *hardware* final (la *SmartNIC Netronome*) no disponga de todas las primitivas necesarias para soportarlo directamente.

4.5. Consideraciones y análisis de la arquitectura

Existen algunas diferencias entre nuestra implementación y la arquitectura presentada en el trabajo de referencia, varias de las cuales ya fueron mencionadas a lo largo del capítulo. En esta sección se reúnen dichas diferencias y se incorporan otras que, hasta este punto, no habían sido discutidas. Las diferencias identificadas se limitan a los aspectos de alto nivel efectivamente descritos en el paper de referencia. Dado que ni el diseño detallado ni la implementación concreta de esa arquitectura son públicos, es posible que existan otras divergencias que no pueden ser determinadas. A continuación se enumeran únicamente las diferencias que pueden establecerse en función de la información disponible.

1. En nuestro trabajo no se implementaron la totalidad de los eventos descritos en el trabajo de referencia. En cambio, se implementaron únicamente los eventos vinculados a *hop latency*. El motivo de este cambio es que a los efectos de este trabajo interesaba simplemente demostrar la posibilidad de implementar algún tipo de notificación de evento.

2. El tamaño de la tabla de hash H_P es menor al tamaño reportado en el paper de referencia. En particular, en este trabajo H_P tiene 2^{18} filas, mientras que en el trabajo de referencia tiene 2^{19}.

   La elección de un tamaño de 2^{18} filas para la tabla de hash H_P responde a una limitación práctica de la organización de memoria en las *SmartNICs* utilizadas. Cada dispositivo divide su memoria externa en tres islas de `EMEM`, de aproximadamente 682 MB cada una. La tabla H_P requiere almacenar, por fila, 12 *buckets* de 204 bytes, lo que implica que una instancia de tamaño 2^{19} superaría la capacidad disponible en una única isla de *EMEM*. En cambio, una tabla de tamaño 2^{18} ocupa aproximadamente 642 MB, lo que permite ubicarla íntegramente dentro de una sola isla de memoria sin necesidad de particionar la estructura.

   Si bien sería posible dividir H_P en múltiples segmentos distribuidos entre distintas islas, a los efectos de este trabajo alcanza con utilizar 2^{18} filas.

3. Los *ring buffers* tienen 2^{17} entradas, en lugar de las 2^{16} que se utilizan en el trabajo de referencia. Este cambio responde a que, dado que redujimos el tamaño de la tabla H_P podemos prevenir posibles desbordes de las estructuras de datos aumentando el tamaño de los *rings*.

4. Se incorporó un *pipeline* de visualización en tiempo real basado en una arquitectura de análisis y almacenamiento centralizado, en lugar del mecanismo de almacenamiento en *Redis* propuesto en el trabajo original. Esta modificación facilita la inspección profunda, consultas históricas y visualizaciones dinámicas de métricas.

5. Se añadieron campos personalizados (`request_id` e `is_response`) con el objetivo de habilitar la correlación entre flujos de solicitud y respuesta. El objetivo es demostrar que la plataforma es extensible y que con la utilización de placas programables se pueden introducir nuevos metadatos necesarios para casos de uso específicos, incluso cuando esos metadatos estén por fuera del soporte de INT.

4.5.1. Escalabilidad de la plataforma

La arquitectura desarrollada se diseñó con el objetivo de ser funcional en un entorno limitado y monolítico -una única *SmartNIC* y un único host de monitoreo-, pero manteniendo compatibilidad con escenarios más exigentes en términos de volumen de tráfico y número de flujos concurrentes. En esta subsección se discute cómo podría escalar la solución en sus distintos componentes, así como los principales cuellos de botella esperables.

En el plano de datos, la capacidad de la *SmartNIC* para procesar tráfico INT a mayor escala está directamente condicionada por los recursos de *hardware* disponibles: tamaño de la tabla de flujos (H_P), profundidad y cantidad de *ring buffers*, número de *microengines* utilizados y ancho de banda de los enlaces. A medida que aumenta la cantidad de flujos activos o el número de nodos INT instrumentados por paquete, crece el costo de las operaciones de agregación y la presión sobre la tabla H_P. En escenarios de mayor carga, sería necesario ajustar parámetros como las políticas de *evicción*, los umbrales de *age-out* o incluso aplicar técnicas de muestreo para mantener la tasa de procesamiento cercana a la tasa de línea.

En lo que respecta al *pipeline* de monitoreo, la solución basada en *Filebeat*, *Logstash* y *Elasticsearch* admite de forma natural una evolución hacia arquitecturas distribuidas. Aunque en este trabajo todos los componentes se ejecutan en un único host, el diseño es compatible con despliegues en los que existan múltiples *Sinks* y múltiples servidores de almacenamiento. En un escenario de mayor escala, cada host asociado a una *SmartNIC* podría ejecutar una instancia local de *Filebeat* que envíe los datos a uno o varios nodos de *Logstash*, mientras que *Elasticsearch* se desplegaría como un clúster con varios nodos de datos y de coordinación. De esta forma, el incremento en el volumen de métricas podría absorberse añadiendo nodos al clúster, sin modificar la lógica de generación ni el formato de los registros.

Desde el punto de vista de la visualización, *Kibana* es capaz de trabajar sobre índices particionados temporalmente (por ejemplo, diarios o mensuales) y distribuidos en varios nodos de *Elasticsearch*. Esto permite mantener tiempos de respuesta aceptables incluso cuando el conjunto de datos crece de forma significativa, siempre que se acompañe con políticas adecuadas de retención, borrado y *rollover* de índices. En este sentido, la estructura de los documentos generados (con campos ya normalizados y preparados para agregaciones) favorece la construcción de paneles que puedan operar sobre volúmenes crecientes de información sin requerir transformaciones adicionales en tiempo de consulta.

En resumen, aunque la implementación concreta descrita en este trabajo se limita a un entorno de laboratorio con recursos acotados, la separación clara entre el plano de datos (*SmartNIC*) y el plano de ingestión, almacenamiento y visualización (stack ELK) habilita una evolución gradual hacia despliegues de mayor escala. Dicha evolución se apoyaría principalmente en la incorporación de *hardware* más capaz (tanto en la *SmartNIC* como en los servidores) y en la distribución de los componentes de ingestión e indexación, manteniendo intactas las interfaces y formatos definidos en esta primera versión de la plataforma.

4.5.2. Limitaciones del entorno

El entorno experimental utilizado presenta una serie de limitaciones inherentes al *hardware* empleado y a las herramientas disponibles. Estas restricciones no impiden el desarrollo del trabajo, pero sí condicionan el alcance de ciertos experimentos y la completitud en terminos de cuantas métricas definidas en la especificación de INT se pueden obtener utilizando estas *SmartNICs* en nodos intermedios. A continuación se detallan las principales limitaciones identificadas:

1. **Capacidad y rendimiento de las placas de red.** Las placas empleadas disponen de una capacidad y velocidad menores en comparación con dispositivos programables más modernos. En el mercado existen placas con mayor ancho de banda, más memoria y velocidades de procesamiento superiores, lo que permitiría escenarios de monitoreo más exigentes o con mayor granularidad temporal.

2. **Limitaciones funcionales del *hardware* frente a soluciones basadas en software.** A diferencia de un *switch* programado completamente en software, las placas utilizadas no soportan la totalidad de las características avanzadas del lenguaje de programación del plano de datos. Como consecuencia, no es posible obtener ciertas métricas que serían valiosas para nuestro caso de uso, especialmente aquellas asociadas a metadatos internos del proceso de reenvío.

3. **Escasa documentación y soporte limitado.** Las placas no cuentan con manuales actualizados ni con soporte activo por parte del fabricante, y la comunidad de usuarios es reducida. Esto dificulta la resolución de problemas, limita el acceso a ejemplos funcionales completos y restringe

la posibilidad de obtener información detallada sobre capacidades internas del *hardware*.

4.6. Conclusiones

El diseño e implementación del *Sink* constituyen un elemento central para una plataforma de monitoreo basada en INT. Procesar paquetes INT de forma continua, mantener el estado asociado a los flujos y generar notificaciones sin alterar el recorrido del tráfico requiere una arquitectura capaz de combinar operaciones ligeras en el plano de datos con tareas de agregación y análisis.

La *SmartNIC* utilizada en este trabajo permite articular estas funciones mediante un *pipeline* que integra la extracción de encabezados en P4 con rutinas ejecutadas en *Micro-C*. La estructura de agregación basada en una tabla hash y la exportación de eventos a través de *ring buffers* permiten separar el procesamiento en la *SmartNIC* del análisis posterior en el *host*.

Sobre esta base, el entorno de monitoreo implementado en el *host* organiza, normaliza y almacena tanto las métricas agregadas como los eventos generados por el *Sink*, habilitando consultas flexibles y visualización estructurada.

Asimismo, la prueba de concepto del nodo intermedio INT aportó evidencia concreta sobre las capacidades y limitaciones del *hardware* utilizado. La evaluación funcional sobre un entorno software demostró que las operaciones propias de un *transit hop* pueden implementarse correctamente, a la vez que permitió identificar restricciones del entorno *Netronome* que condicionan su implementación directa sobre la *SmartNIC*.

La plataforma resultante ofrece un camino claro para instrumentar dominios INT completos, integrando captura, procesamiento y análisis de manera coherente y extensible.

En conjunto, este capítulo establece los fundamentos técnicos de la plataforma desarrollada y deja delineado el alcance real de su diseño. Sobre esta base, en el siguiente capítulo se presenta el producto desde la perspectiva del usuario, incluyendo sus capacidades, visualizaciones y casos de uso habilitados por la plataforma. Y más adelante, en el Capítulo 5, se introduce la evaluación del sistema, donde se analiza su desempeño y la precisión de las métricas recolectadas.

Capítulo 5

Evaluación experimental

En este capítulo se presenta la evaluación del INT *Sink* y se organiza de la siguiente manera:

- En la Sección 5.1 se describe el entorno experimental, incluyendo la configuración del *host* generador, del *host Sink* y la traza de tráfico de red utilizada.
- La Sección 5.2 presenta los resultados de rendimiento en términos de *throughput* y la Sección 5.3 los resultados del rendimiento en términos de latencia en las operaciones principales.
- La Sección 5.4 analiza la precisión de las métricas INT recolectadas por el *Sink*.
- Finalmente, en la sección 5.5 se valida el correcto funcionamiento de la capa de visualización.

5.1. Entorno de evaluación

Para la evaluación experimental se utilizaron dos hosts físicos. Cada host ejecuta Linux con kernel 4.18.0-15-generic y está equipado con un procesador AMD Ryzen 7 7700 (8 núcleos / 16 hilos) y 32 GB de memoria RAM. Ambos sistemas incorporan una tarjeta *SmartNIC Netronome Agilio 4000 CX* de doble puerto 10 GbE.

Los dos hosts se conectaron directamente mediante enlaces punto a punto. En particular, los puertos físicos `p0` de ambas tarjetas se conectaron entre sí, al igual que los puertos `p1`.

5.1.1. Generación de tráfico

Para la generación del tráfico se utilizó una traza pública del conjunto de datos propuesto en [51], que recoge capturas de tráfico en un entorno de red

real. A partir de uno de los archivos `pcap` del dataset se construyó una nueva traza instrumentada con INT.

El preprocesamiento de la traza se realizó mediante un *script* en Python que aplica las siguientes transformaciones:

- Se filtran únicamente los paquetes que contienen encabezado IP y se truncan los *frames* Ethernet a 64 bytes, de modo de maximizar la tasa de llegada de paquetes en las pruebas de *throughput*.

- Por cada paquete original se generan dos paquetes, interpretados como *request* y *response*, invirtiendo direcciones IP y puertos de transporte (TCP o UDP) en la respuesta.

- A cada paquete se le antepone un encabezado de metadatos de aplicación que incluye un identificador de flujo de 16 bits y un bit que indica si se trata de un paquete de respuesta.

- Sobre cada paquete (*request* y *response*) se inyectan encabezados y *stacks* INT.

- Los valores del campo `node_id` (que termina identificando los caminos INT) se asignan dependiendo de si el paquete corresponde a una *request* o una *response*. Por simplicidad se asume que a ambos lados del *Sink* existen la misma cantidad de saltos *INT*, como muestra la figura 5.1, y que los mismos insertan metadatos solo cuando viajan en dirección al *Sink*. De ese modo, lo que se busca simular con la traza es que: (i) llegan paquetes que pasaron por los *hops* 1 y 2, son procesados por el *Sink* y salen sin metadatos INT; (ii) cuando la request llega a su destino, y vuelve el paquete correspondiente a la response, el mismo pasa por los *hops* 4 y 3 (en ese orden), para luego ser procesados en el *Sink*. A los efectos de la evaluación experimental, estos paquetes de *request* y *response* se generan en el mismo archivo `pcap` pues son enviados a una única interfaz del *Sink*, como se explicará mas adelante.

- Los valores concretos de los campos de metadatos INT (por ejemplo, latencia de salto, ocupación de cola o marcas de tiempo de ingreso/egreso) se generan de forma sintética siguiendo distribuciones configurables.

El resultado de este proceso es un nuevo archivo `pcap` en el que cada paquete del dataset original se transforma en un par *request*/*response* con encabezados INT insertados y simulando llegar al *Sink* desde caminos en sentido inverso.

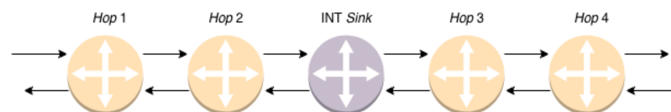


Figura 5.1: Caminos INT a cada lado del *Sink*

5.2. *Throughput*

Antes de analizar el rendimiento del INT *Sink*, se realizó una prueba de referencia destinada a cuantificar el *throughput* alcanzable bajo un programa P4 minimalista. Esta prueba establece un punto de comparación necesario para evaluar el costo añadido por la instrumentación INT y por las operaciones adicionales implementadas en el *Sink*.

El experimento se realizó utilizando ambos *hosts* físicos y sus respectivas *SmartNICs*. El esquema de reenvío configurado fue el siguiente:

- En el Host A, el programa P4 reenvía cualquier paquete recibido por `vf0_0` hacia `p1`, y reenvía todo paquete recibido por `p1` hacia `vf0_1`.
- En el Host B, inicialmente se cargó un programa P4 extremadamente simple que reenvía todo paquete recibido por `p1` nuevamente hacia `p1`.

MoonGen se ejecutó sobre el Host A, transmitiendo a tasa de línea (10Gbps) una traza de 929,940 paquetes hacia `vf0_0`, llegando a una velocidad de 7.658 millones de paquetes por segundo dado el tamaño de dichos paquetes. El flujo P4 encadena así ambos *pipelines* y retorna los paquetes al Host A. En la Figura 5.2 se muestra un diagrama de esta configuración.

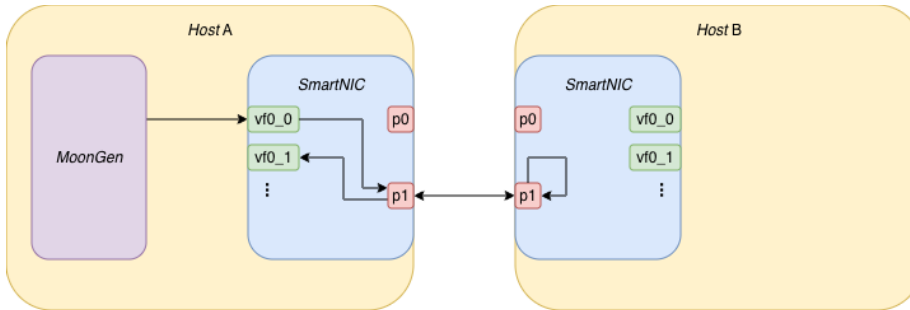


Figura 5.2: Configuración para prueba de *baseline*.

Con el programa básico en el Host B, únicamente 810,837 de los 929,940 paquetes enviados alcanzan el *pipeline* de dicho host a través de `p1`. Los mismos 810,837 paquetes son reenviados de vuelta al Host A.
La pérdida de paquetes asciende a aproximadamente 12.8 %:

$$1 - \frac{810{,}837}{929{,}964} \approx 0,128$$

Este resultado es especialmente relevante dado que:

1. No se están realizando operaciones INT, transformaciones de encabezados, acceso a memoria compartida, ni operaciones de cómputo intensivo en los MEs.

2. El *forwarding* es puramente determinístico, consistente en dos reglas de coincidencia exacta.

3. El generador transmite a línea; por lo tanto, cualquier reducción observada corresponde exclusivamente al comportamiento del firmware básico de reenvío.

Se observa entonces que el reenvío a tasa de línea no está garantizado incluso bajo carga mínima.

Se repitió la misma prueba reemplazando en el Host B el programa trivial por el programa completo del *Sink*. Bajo esta configuración, el número de paquetes que siquiera ingresan al *pipeline* del Host B se reduce drásticamente: solo 171,579 paquetes de los 929,940 transmitidos alcanzan el bloque de reenvío del *Sink*. Estos mismos 171,579 paquetes son los que finalmente retornan al Host A.

Un aspecto destacable es que esta reducción no se debe a pérdidas dentro del *pipeline* P4, sino a que los paquetes no ingresan al *pipeline* en primer lugar. En otras palabras, la limitación ocurre un paso más “arriba”, en componentes que determinan cuántos paquetes pueden ser absorbidos por el *pipeline* antes de iniciar la etapa de *ingress*.

Si bien estos resultados muestran degradaciones marcadas (especialmente bajo el firmware del *Sink*), es importante señalar que el trabajo de referencia [19] reporta procesamiento a tasa de línea sobre dispositivos *Netronome Agilio CX*. Esto sugiere que el comportamiento observado en esta prueba no debe interpretarse como una limitación inherente del *hardware*, sino como una caracterización específica del entorno y la configuración utilizados en este experimento.

En consecuencia, el *baseline* presentado aquí debe entenderse como una medición empírica del sistema tal como fue configurado para este trabajo. Las evaluaciones subsiguientes del INT *Sink* se construyen tomando este punto de referencia como línea base para analizar el impacto del procesamiento adicional introducido por las funcionalidades INT.

5.2.1. *Throughput* vs cantidad de nodos/instrucciones INT

Para evitar la pérdida de paquetes antes del *ingress* observada en las pruebas iniciales, se modificó el entorno experimental de modo que el tráfico fuese generado desde el mismo host donde corre el *Sink*. En esta configuración, *MoonGen* transmite hacia `vf0_0`, donde el *Sink* procesa los paquetes y luego los reenvía a través de `vf0_3`, interfaz en la que podemos capturar tráfico.

Este esquema garantiza que todos los paquetes generados ingresan efectivamente al *pipeline* P4.

Un efecto observado es que, al ejecutar *MoonGen* en el mismo host, la tasa de transmisión alcanzada por DPDK es menor que cuando se transmite desde otro equipo. Se observa que la tasa máxima a la que *MoonGen* puede enviar los paquetes se ve limitada por la capacidad de procesamiento del firmware cargado en la *SmartNIC* de la PC en la que se ejecuta. Por tanto, en todas las configuraciones evaluadas no se registró pérdida de paquetes: el *Sink* reenvía el 100 % de los paquetes recibidos.

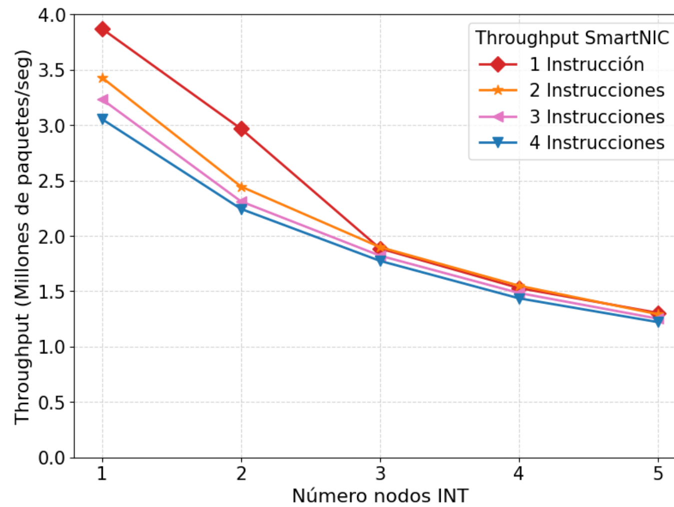


Figura 5.3: *Throughput* de paquetes procesados por la *SmartNIC*.

La Figura 5.3 muestra el *throughput* obtenido para distintas combinaciones de nodos INT (1 a 5) y número de instrucciones (1 a 4). Se observa una disminución progresiva del *throughput* al incrementar la cantidad de nodos o instrucciones, lo cual refleja el costo adicional del procesamiento INT. Los valores oscilan desde aproximadamente 3.8 Mpps (1 nodo, 1 instrucción) hasta 1.2 Mpps (5 nodos, 4 instrucciones). Para dimensionarlo en términos de tasa de bits, dependiendo del tamaño del paquete (naturalmente los paquetes con más instrucciones y nodos INT tienen más bits), estos rangos van desde los 1600 a los 2700 Mbps.

Con el objetivo de encontrar si el cuello de botella está relacionado con la tasa de bits o con la tasa de paquetes, se experimentó enviar una traza cuyos paquetes ocupan el máximo tamaño (MTU). En este caso se observa como se alcanza sin problemas la tasa de linea de 10 Gbps, a una tasa de 0.833 millones de paquetes por segundo (la traza contiene metadatos de 5 nodos INT con 4 instrucciones). Esto nos indica que si bien el INT *sink* puede procesar paquetes a la tasa de línea nominal de la *smartNIC*, hay un cuello de botella en el número de paquetes por unidad de tiempo que puede procesar. Dicho de otro modo, esa tasa de línea nominal se puede alcanzar siempre y cuando los paquetes no sean demasiado pequeños. Este punto se retoma más abajo.

En ese sentido, la Figura 5.3 refleja la máxima tasa de paquetes por segundo para cada configuración, dado que se utilizó una traza que achica los paquetes del *dataset* original a su mínimo tamaño, con el fin de estresar al sistema tanto como sea posible.

5.2.2. Posibles causas de degradación de *throughput*

Como se observa tanto en la prueba de *baseline* como en las mediciones del *Sink* INT, el *throughput* máximo alcanzado se ve limitado en términos de paquetes por segundo, incluso bajo configuraciones mínimas de procesamiento. Este comportamiento impide, para paquetes en cierto rango de tamaños, alcanzar los 10 Gbps nominales que la *SmartNIC* debería soportar y coincide con la degradación reportada al incrementar la complejidad del procesamiento INT.

Si bien, como se mencionó anteriormente, el *hardware* no impone necesariamente este límite, ya que existen resultados publicados que muestran procesamiento a tasa de línea para esta misma familia de dispositivos, varios factores pueden contribuir a la reducción observada en este entorno experimental. A continuación se resumen las causas posibles.

Configuración del entorno y del plano PCIe. El rendimiento puede verse condicionado por parámetros de bajo nivel del entorno de ejecución, incluyendo:

- configuración de las VF,
- profundidad de colas y número de *descriptors*,
- créditos internos asociados al *pipeline* de recepción,
- comportamiento del driver y su interacción con el plano PCIe.

Estos parámetros afectan la velocidad a la que los paquetes pueden ser absorbidos por el *pipeline* antes de iniciar la etapa de *ingress*.

Sobrecarga asociada al envío a *host*. Se señala en el soporte de *Netronome* que cuando el tráfico debe pasar del *wire* a la *SmartNIC*, luego al *host*, y finalmente volver a la *SmartNIC* para ser reenviado, el volumen total de trabajo que recae sobre el dispositivo se duplica. Este patrón coincide con algunos de los experimentos realizados en este trabajo, donde el camino *wire* → VF → *pipeline* → VF → *wire* incrementa significativamente la carga sobre los caminos *DMA* y sobre los microengines involucrados [42].

Tamaño reducido de los paquetes. El uso de paquetes cuyo tamaño fue intencionalmente reducido agrava la situación, ya que el *overhead* por paquete es relativamente alto en relación con la cantidad de datos procesados. En el soporte técnico de *Netronome* se advierte que el rendimiento en Mpps tiende a degradarse con paquetes pequeños, y que tasas superiores pueden alcanzarse únicamente con tamaños mayores [42].

Distribución de carga entre microengines. El firmware de la *SmartNIC* utiliza un sistema de créditos para distribuir el trabajo entre los microengines disponibles. Aunque este mecanismo evita la sobrecarga de un único *microengine*, también implica que:

- la asignación de paquetes a islas y contextos no es determinista,
- un mismo flujo no necesariamente se procesa en la misma isla,
- variaciones en la distribución de carga pueden introducir pérdida si todos los contextos registrados se encuentran ocupados.

El soporte técnico también menciona que, bajo ciertas condiciones, un único BP (*buffer pool*) puede quedar saturado, limitando la tasa de ingesta [42].

Particularidades del firmware y del proceso de carga. Diferencias en la forma en que se compila y carga el diseño (incluso sin INT) pueden influir en:

- cuánta memoria se reserva para estructuras internas,
- qué microengines se asignan a la tarea de recepción,
- cómo se configuran los BP entries para balanceo de carga.

Esto puede generar degradación incluso en programas de *forwarding* aparentemente triviales.

Decisiones de diseño del *Sink*. Además de los factores anteriores, la implementación del *Sink* introduce procesamiento adicional en *Micro-C*: clasificación de flujos, acceso a memoria compartida, actualización de estructuras y exportación de eventos. Si bien estas operaciones no explican por sí solas la pérdida en el *baseline*, sí incrementan el costo por paquete para el *firmware* del *Sink*.

En conjunto, estos factores ofrecen un marco razonable para interpretar la degradación observada. El límite en Mpps no corresponde a una restricción fundamental del *hardware*, sino al resultado de una combinación de condiciones del entorno experimental, características del *firmware* utilizado y decisiones de diseño propias de esta implementación.

5.2.3. Ensayos de optimización del *pipeline*

En esta sección se discute sobre algunos de los caminos explorados para mejorar el rendimiento del programa.

Optimizaciones en P4 En esta etapa del *pipeline* lo que se buscó fue optimizar el bloque *MyParser*, reduciendo la cantidad de estados y transiciones. Disminuir la cantidad de estados ayuda a mejorar la velocidad de procesamiento pero no se observó una mejora significativa. En contraposición, limitar los estados hace que el *Parser* esté acotado ante las distintas combinaciones posibles de paquetes INT que puedan llegar, por lo que se descarto rápidamente esta opción. Otra optimización que si arrojó resultados notorios fue la de no utilizar el mecanismo de `pop_from_stacks()` y pasar a trabajar con los índices de la pila *stack_ element_ t*. Esto se debe a que, para los *FPC*, la acción implica

un esfuerzo mayor comparado con otras operaciones básicas. A pesar de esto, al igual que para el caso anterior, esto limita las capacidades del *Parser* y hace que quede *hardcodeado* a un tipo de paquete INT que no tiene porque ser siempre el mismo.

Optimizaciones en Micro-C *Netronome* ofrece dentro de sus manuales, una sección dedicada a la optimización del código C, de las cuales se exploraron la mayoría. Algunos de estos intentos de optimización incluyeron:

- Reestructuración de la organización del código: cada vez que se genera una rama en el código, se coloca aquel código que tiene más probabilidades de ser ejecutado inmediatamente después y así garantizar un procesamiento secuencial del código sin saltos.
- Reducción de accesos a memoria global: los accesos a la tabla H_p son altamente costosos en comparación con los registros o memoria local de cada *microengine*. Los accesos a memoria local se dan entre 1 y 3 ciclos de reloj mientras que el de memoria global entre 150 y 500. Se probó el *caching* de entradas accedidas frecuentemente. No se presentaron mejoras concluyentes.
- Mover código fuera de secciones controladas por los semáforos: existen algunas porciones reducidas de código que no dependen de accesos a recursos compartidos, por lo que se se movieron fuera de zonas de mutua-exclusión para que cada *microengine* e hilo pueda hacer procesamiento adicional en paralelo. Esto tampoco arrojó cambios significativos en las pruebas de rendimiento.

5.3. Latencia del procesamiento

Latencia introducida por las principales operaciones Debido a que P4 no ofrece la capacidad de medir cuanto tiempo consume el procesamiento fuera del entorno de *Micro-C*, las mediciones que se proponen son las siguientes:

1. Cuando se actualiza una entrada sobre H_P.
2. Cuando se realiza una evicción.

Para obtener mediciones que representen fielmente la realidad se utilizo una traza con 5 nodos y 4 instrucciones INT, de la cual se mandaron 4095 paquetes, los cuales constituyen 2097 flujos distintos y una cantidad de paquetes por flujo variante.

Latencia en actualización En la tabla 5.1 se puede ver como el tiempo que toma en actualizar la tabla H_p se triplica cuando el flujo tiene más de un paquete (lo esperado en escenarios realistas y no de pruebas). Esto muestra que los cálculos del promedio de las métricas son la principal fuente de latencia en

Experimentos	Cantidad	Min	Max	Media	Desv. Est.
Primera inserción en H_p	2097	5.16 µs	8.01 µs	5.89 µs	0.32 µs
Act. de entrada en H_p	1998	**1.78 µs**	20.25 µs	**18.85 µs**	0.44 µs
Flujo completo	4095	5.0 µs	**20.27 µs**	12.05 µs	**6.53 µs**
Evicción	1240	5.5 µs	8.37 µs	6.77 µs	0.52 µs

Tabla 5.1: Resumen estadístico de los experimentos.

esta parte del procesamiento del paquete, lo cual es esperado. Los cálculos del promedio en nuestras pruebas consisten en una resta, suma, multiplicación y división por métrica. A modo de experimentación, se probó con simplemente una suma (lo cual delegaría al *host*, en alguna de sus etapas, hacer el promedio haciendo simplemente una división) y se obtuvieron tiempos muy similares a los obtenidos en la primera inserción en H_P.

Latencia en evicción Para estas pruebas forzamos a que hubieran colisiones bajando la cantidad de entradas en H_p a 2^{11} y 2 *buckets*. Se generaron 1240 colisiones donde las filas de la tabla estaban llenas. El tiempo medio que lleva escribir un *bucket* desde H_p a su correspondiente *ring buffer* es similar al de la actualización por primera vez en la tabla. Esto es similar a los reportado en [19].

5.4. Precisión de las métricas INT

Prueba unitaria Como verificación inicial, se transmitió un único paquete INT y se inspeccionaron los campos almacenados en *Elasticsearch*. Se comprobó que cada metadato reportado, incluyendo `node_id`, `hop_latency`, `queue_occupancy` y `egress_interface_tx_utilization`, para todos los saltos INT, coincidía exactamente con los valores embebidos en la traza. Esta prueba confirma el correcto funcionamiento del *pipeline* de extracción, agregación y exportación para el caso más simple. A continuación se presenta una versión escalada de esta prueba, en la que se mide la precisión para un volumen más grande de paquetes.

Comparación de distribuciones Para evaluar la precisión con la que el *Sink* reconstruye las métricas INT, se diseñó un experimento orientado a contrastar los valores reportados por el *Sink* con la traza original. A diferencia de las trazas empleadas en las mediciones de rendimiento, esta evaluación utiliza un conjunto sintético compuesto por 100.000 paquetes pertenecientes a flujos distintos.

La elección de flujos independientes responde a que la arquitectura del *Sink* agrega métricas por flujo antes de su exportación. Para medir exactitud a nivel de paquete (como exige esta evaluación) es necesario evitar dicha agregación. El uso de flujos distintos garantiza que cada paquete sea tratado de manera independiente, preservando toda la información INT exportada.

Como se mencionó, la comparación se realiza entre dos fuentes de datos:

- **Valores de referencia**: métricas generadas y embebidas directamente en el tráfico INT sintetizado.
- **Valores en *SmartNIC***: métricas exportadas por la *SmartNIC* y entregadas al *pipeline* de visualización.

El objetivo es contrastar la distribución de ocurrencias de una métrica INT seleccionada para un nodo individual. La discrepancia entre ambas distribuciones se cuantifica mediante la siguiente métrica de pérdida:

$$\text{Tasa de pérdida} = 1 - \frac{\#\text{Ocurrencias observadas en la } \textit{SmartNIC}}{\#\text{Ocurrencias en valores de referencia}}.$$

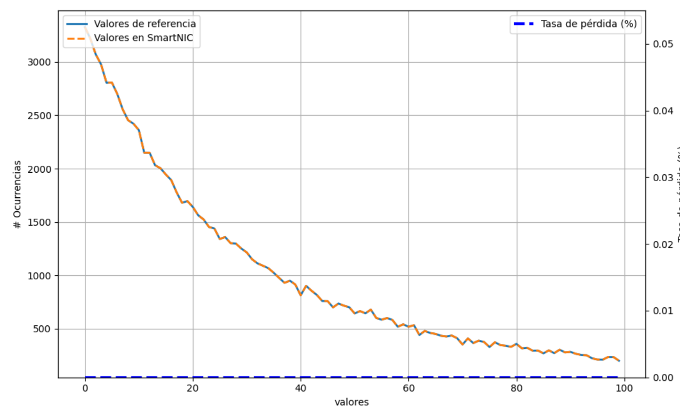


Figura 5.4: Comparación de las distribuciones de utilización del enlace TX en el primer hop INT.

La figura 5.4 muestra la comparación de distribuciones de utilización del enlace TX de la interfaz de salida (una de las métricas INT recolectadas) para el primer salto INT. Los valores observados en dicha figura corresponden a una traza enviada a 2,8 millones de paquetes por segundo, sin pérdida de paquetes.

En la figura se muestra que la tasa de pérdida es 0 % para todos los valores. Es decir, en un caso en el que no hay pérdida de paquetes el valor reportado por la *SmartNIC* al *host* es idéntico al valor contenido en la traza, para todos los casos.

La utilización TX del primer salto se presenta únicamente con fines ilustrativos; la misma presición se observa para todas las métricas y en todos los saltos INT.

Este experimento permite demostrar el correcto funcionamiento de la plataforma en términos de precisión de los valores reportados.

Esta prueba fue llevada a cabo con *MoonGen* corriendo en el mismo host que el *Sink*. Esto hace que la tasa a la que envía *MoonGen* sea la tasa más alta que la plataforma puede soportar antes de descartar paquetes.

Con el objetivo de forzar una tasa de paquetes más alta de la que la plataforma puede soportar, reprodujimos la traza con *MoonGen* desde la otra PC de nuestro *setup*, haciéndole llegar al *Sink* los paquetes a través de la interfaz p1. En particular, se envío la traza a una velocidad de 3.655 millones de paquetes por segundo. Los resultados se pueden observar en la figura 5.5. La tasa de pérdida se debe a pérdida de paquetes ocasionada por recibir los mismos a una velocidad más alta de la que el firmware los puede procesar. De los 100.000 paquetes enviados, solamente 62.319 ingresan al *pipeline* de procesamiento del *Sink*. Y a esos 62.319 paquetes corresponden los valores observados en la figura. Si bien la plataforma reporta al *host* correctamente todos los valores de los paquetes que ingresan al *pipeline*, hay paquetes que ni siquiera llegan a ingresar cuando se trabaja con altas tasas, como se reportó en la sección anterior.

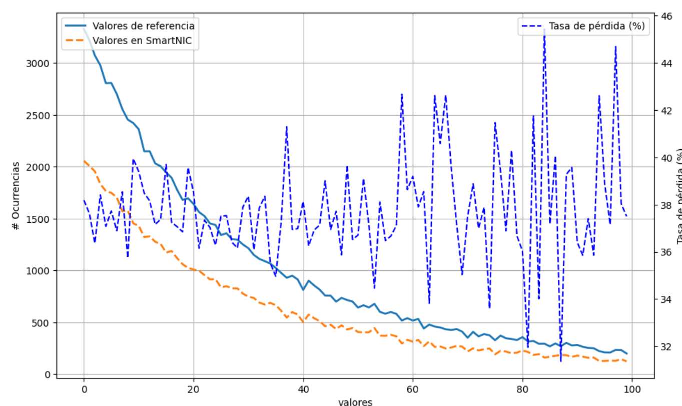


Figura 5.5: Comparación de las distribuciones de utilización del enlace TX en el primer hop INT bajo condiciones de degradación de *throughput*.

Los resultados obtenidos permiten verificar que la lógica de reconstrucción y exportación implementada en el *Sink* opera de manera correcta. En condiciones sin pérdida de paquetes, las métricas reportadas por la *SmartNIC* coinciden exactamente con los valores embebidos en la traza, reproduciendo su distribución sin discrepancias para todos los nodos y para todas las métricas evaluadas.

Cuando la tasa de envío supera la capacidad de procesamiento del firmware, la degradación observada no se debe a errores en la reconstrucción de métricas, sino a que una fracción de los paquetes no llega a ingresar al *pipeline* de procesamiento. En esos casos, la *SmartNIC* reporta únicamente las métricas asociadas a los paquetes efectivamente procesados. Por lo tanto, las diferencias entre ambas distribuciones bajo condiciones de sobrecarga reflejan pérdida previa a la etapa de extracción de metadatos, y no imprecisiones en el mecanismo de recolección.

Habiendo dicho esto, y en línea con las conclusiones de la sección anterior, se observa que ante tasas elevadas la plataforma se degrada y, en un sentido más holístico, la precisión final sí se ve afectada. Este impacto no proviene de una lógica incorrecta, sino del hecho de que los paquetes descartados nunca

atraviesan el *Sink*, por lo que su información INT no puede ser registrada ni considerada en la comparación.

5.5. Validación de la capa de visualización

Como parte del proceso de evaluación experimental, se validó el correcto funcionamiento de la capa de visualización implementada con ELK. Para ello, se desarrollaron *dashboards* que permiten inspeccionar la ingestión de datos desde la *SmartNIC*. Las Figuras 5.6 y 5.7 muestran ejemplos de estos *dashboards*.

La Figura 5.6 muestra un *dashboard* diseñado para obtener una caracterización general del tráfico instrumentado. Entre las métricas presentadas se encuentran: número total de flujos observados, cantidad acumulada de paquetes, promedio de nodos INT por flujo, caminos y su latencia promedio, y distribución de flujos y paquetes por camino INT.

La Figura 5.7, por su parte, muestra un tablero orientado a un nodo INT específico. El nodo observado se selecciona por medio de un *dropdown*. Cada visualización resume la evolución temporal de distintas métricas de ese salto: latencia de *hop*, utilización de la interfaz de egreso y ocupación de cola.

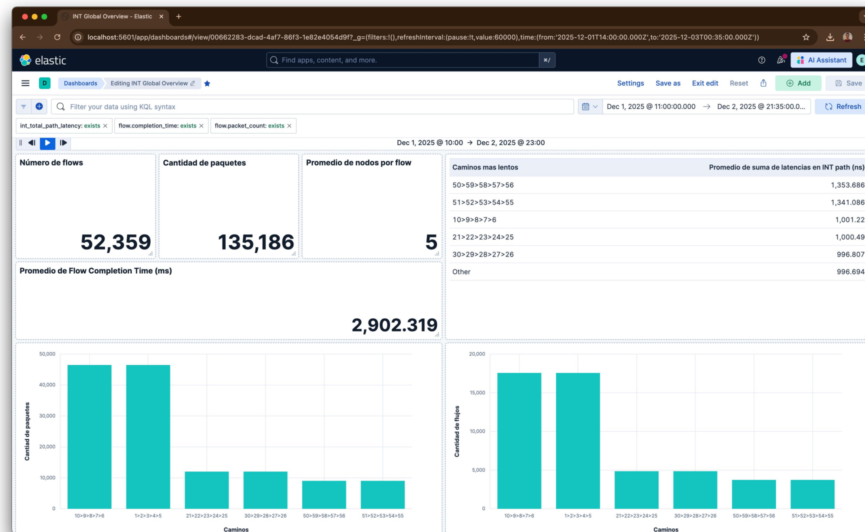


Figura 5.6: *Dashboard* de visión global.

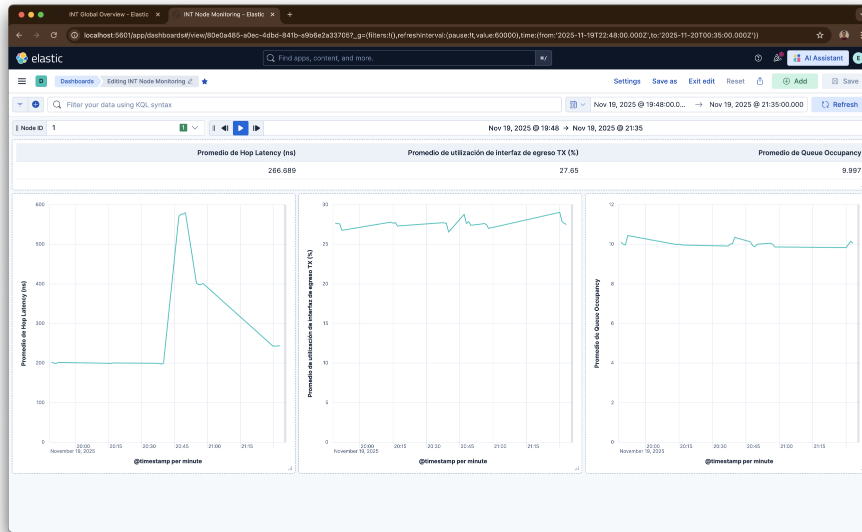


Figura 5.7: *Dashboard* por nodo INT.

Los valores mostrados en los *dashboards* fueron contrastados tanto con los registros almacenados en los archivos `ndjson` como con los valores originales de la traza, lo que permitió confirmar la consistencia del proceso completo de recolección, exportación y visualización de los metadatos INT.

5.6. Conclusiones

La evaluación experimental permitió analizar el desempeño del INT *Sink* desarrollado y su comportamiento bajo distintos escenarios de carga.

En términos de precisión, los resultados muestran que el sistema reporta métricas INT de forma correcta siempre que no exista pérdida de paquetes. Bajo estas condiciones, las distribuciones observadas en el *host* coinciden con la traza original. En situaciones con degradación, la pérdida de paquetes impacta naturalmente sobre la exactitud de las métricas reportadas, ya que los metadatos asociados a los paquetes descartados no alcanzan el plano de análisis.

Por otro lado, a través del uso de *dashboards*, se realizó la validación de los datos que llegan a la capa de visualización. Esto permitió corroborar, una vez más, la precisión de la plataforma implementada.

En cuanto al *throughput*, los experimentos evidencian que la capacidad de procesamiento del sistema está fuertemente vinculada a la tasa de paquetes por segundo. Si bien es posible alcanzar tasas de línea en términos de bits por segundo, la plataforma no logra sostener la misma tasa cuando la exigencia se expresa en términos de paquetes.

Respecto a la latencia, los tiempos reportados para la evicción son comparables con los del articulo de referencia [19]. Lo mismo sucede con la primera inserción de un flujo en la tabla H_p, el cual muestra un tiempo incluso más bajo que el de la evicción. La principal diferencia con respecto al articulo de referencia es cuando se actualizan las entradas en la tabla, viéndose aquí como se triplica en tiempo y siendo un factor importante que puede explicar parte de los resultados en pruebas de rendimiento. Esta degradación en la latencia de la actualización del flujo se explica por el cálculo de promedios en las métricas. Aquí hay lugar a mejoras, como por ejemplo no calcular el promedio a medida que llegan paquetes, si no que ir sumando los valores que lleguen y luego cuando ocurra una evicción (sincrónica o asincronía), simplemente dividir el valor de la métrica por la cantidad de paquetes.

Además, se exploraron distintas estrategias orientadas a mejorar el rendimiento del sistema, aunque ninguna de ellas produjo mejoras concluyentes.

Finalmente, se presentaron posibles factores que pueden tener impacto sobre el rendimiento de la plataforma. Algunos de ellos relacionados con la implementación concreta de este trabajo, y otros de ellos relacionados a características como el entorno de experimentación y el funcionamiento de las *smartNICs* utilizadas.

Capítulo 6

Conclusiones y Trabajo Futuro

Este proyecto tuvo como objetivo reproducir, extender y evaluar una plataforma de monitoreo basada en INT tomando como referencia la arquitectura propuesta en *"A SmartNIC-Accelerated Monitoring Platform for In-band Network Telemetry"* [19]. Sobre esa base se desarrollaron múltiples componentes: un INT *Sink* operativo en una *SmartNIC Netronome Agilio CX*, un *pipeline* de visualización completo basado en ELK, extensiones para incorporar metadatos a nivel de aplicación, y una prueba de concepto de nodo INT intermedio. Este capítulo resume los resultados obtenidos, revisa las decisiones tomadas, describe las dificultades encontradas y presenta líneas de trabajo futuro.

6.1. Evaluación de los resultados alcanzados

En términos generales, la plataforma implementada demostró un funcionamiento correcto en lo que respecta a la recolección y exportación de métricas INT. Bajo condiciones controladas, en las que no se producen pérdidas previas de paquetes, las métricas reconstruidas coinciden con los valores originales embebidos en las trazas utilizadas para las pruebas. Esto indica que el procesamiento en la *SmartNIC* -incluyendo la extracción de metadatos, su agregación y su exportación hacia el *host*- opera de acuerdo con el diseño previsto.

Por otra parte, las mediciones de rendimiento mostraron que el *throughput* alcanzable se encuentra limitado en la configuración experimental utilizada. Si bien la plataforma logra procesar varios millones de paquetes por segundo de manera estable, no se alcanzaron las tasas máximas teóricas del *hardware* en escenarios de alta carga, y el rendimiento es menor al reportado en el *paper* de referencia. Este comportamiento refleja restricciones propias del entorno y de las condiciones de operación empleadas, así como posibles decisiones subóptimas en la implementación llevada a cabo en este trabajo. Aun así, el desempeño obtenido permite validar el funcionamiento integral del sistema y sentar una

base sólida para explorar optimizaciones futuras.

6.2. Comparación entre objetivos planteados y resultados alcanzados

Objetivos alcanzados.

- Implementación de un INT *Sink* funcional en la *SmartNIC*, incluyendo extracción de metadatos, agregación por flujo y detección de eventos.
- Exportación de metadatos mediante *ring buffers* hacia el host y construcción del flujo completo de transferencia desde la *SmartNIC* hasta el sistema de análisis.
- Diseño e integración de un *pipeline* basado en ELK para la ingestión, normalización, indexación y visualización de métricas y eventos INT.
- Desarrollo de extensiones para soporte de *application metadata*, habilitando casos de uso orientados a trazabilidad y correlación de solicitudes.
- Implementación funcional de un nodo INT intermedio (*transit hop*) en un entorno software (BMv2).
- Generación de un entorno experimental reproducible, acompañado de herramientas y configuraciones documentadas.

Elementos fuera de alcance o no aplicables.

- Implementación del nodo INT intermedio en la *SmartNIC*. La exploración de su viabilidad se llevó a cabo y permitió concluir que, debido a restricciones del SDK y del modelo de programación disponible, la implementación no es factible en esta plataforma. Fue determinado como técnicamente no realizable.

6.3. Contribuciones

- Publicación abierta de la totalidad del código fuente utilizado en la implementación de la plataforma INT, junto con un entorno experimental completamente reproducible, incluyendo scripts, configuraciones y trazas de prueba.
- Extensión de la arquitectura INT de referencia mediante la incorporación de metadatos de aplicación, habilitando la correlación entre solicitudes y respuestas y ampliando los casos de uso posibles sobre el modelo original.
- Diseño e integración de un *pipeline* de análisis completo basado en ELK, responsable de la ingestión, normalización, indexación y visualización de métricas y eventos INT.

- Caracterización empírica del comportamiento real de la *SmartNIC* en el entorno del *Smartlab*, aportando información útil para futuros trabajos y experimentos sobre esta plataforma.

6.4. Posibles casos de uso

La plataforma ofrece un conjunto de datos suficientemente rico como para habilitar una variedad de aplicaciones. Sin realizar una evaluación completa sobre cada una, se mencionan a continuación algunos escenarios donde este tipo de información podría resultar útil:

- Análisis de rutas y desempeño extremo a extremo: permite estudiar cómo cambian las rutas INT a lo largo del tiempo y cómo varían sus latencias agregadas.
- Detección temprana de congestión: la ocupación de cola y la latencia por salto pueden emplearse como indicadores para alertar sobre condiciones anómalas en la red.
- Diagnóstico de problemas operativos: la correlación entre *hop latency*, *queue occupancy* y utilización de enlaces facilita detectar dónde se origina un aumento de tiempo extremo a extremo.
- Optimización de aplicaciones: los metadatos de aplicación permiten identificar flujos con tiempos de respuesta elevados y estudiar su comportamiento dentro del dominio INT.
- Entrenamiento de modelos de aprendizaje automático: el conjunto de métricas INT puede utilizarse como entrada para modelos de predicción de congestión, clasificación de tráfico o detección de anomalías.

Estos posibles casos de uso están estrechamente relacionados con las posíbles líneas de trabajo a futuro presentadas a continuación.

6.5. Líneas de trabajo a futuro

Algunas líneas de trabajo a futuro incluyen la optimización del *Sink* para mejorar el *throughput* y alcanzar tasas de línea incluso con paquetes pequeños, la revisión de la estrategia de almacenamiento de metadatos de *request* y *response* para habilitar el seguimiento de múltiples paquetes por flujo, y la exploración del uso de los datos recolectados por la plataforma como insumo para algoritmos de aprendizaje automático orientados al análisis del tráfico y a la gestión inteligente de recursos.

Las extensiones incorporadas a la plataforma permiten imaginar su aplicación en entornos más amplios, como despliegues distribuidos o arquitecturas modernas de microservicios [43]. En particular, la disponibilidad de `request_id`

y la reconstrucción explícita del camino INT a través de los `node_id` habilitan formas de análisis que trascienden el monitoreo puramente de red.

El uso de identificadores de solicitud permitiría correlacionar paquetes pertenecientes a una misma operación lógica, posibilitando estudios de trazabilidad a nivel de aplicación y la identificación de puntos donde se introducen demoras sobre el recorrido de una petición. Del mismo modo, la información de rutas efectivamente transitadas (obtenida a partir del orden en que aparecen los `node_id`) ofrece una visión detallada de los caminos reales dentro de la infraestructura, útil para detectar desvíos, cambios inesperados o variaciones de latencia asociados a distintos tramos.

En conjunto, estas capacidades abren la puerta a futuras implementaciones orientadas al diagnóstico y optimización en sistemas distribuidos.

6.6. Conclusión final

El proyecto logró construir y validar una plataforma INT funcional, extensible y capaz de procesar tráfico a varios millones de paquetes por segundo. Además de reproducir los componentes esenciales de la arquitectura de referencia, el trabajo logró extenderla. Los resultados obtenidos demuestran la viabilidad de emplear *SmartNICs* de bajo costo para monitoreo INT y proporcionan una base técnica sólida para investigaciones futuras en correlación de métricas, análisis avanzado y sistemas de monitoreo basados en telemetría embebida.

Bibliografía

[1] Carolyn Jane Anderson et al. "NetKAT: Semantic foundations for networks". En: 2014. DOI: 10.1145/2535838.2535862.

[2] Giuseppe Bianchi et al. "OpenState: programming platform-independent stateful openflow applications inside the switch". En: (abr. de 2014). DOI: 10.1145/2602204.2602211.

[3] Pat Bosshart et al. "P4: programming protocol-independent packet processors". En: *SIGCOMM Comput. Commun. Rev.* (jul. de 2014). DOI: 10.1145/2656877.2656890.

[4] Belen Brandino. *Dispositivos de red programables*. Tesis de grado. Uruguay: Universidad de la República, Facultad de Ingeniería, 2022.

[5] Alisha Cecil. *A Summary of Network Traffic Monitoring and Analysis Techniques*. Inf. téc. CSE 567M: Computer Systems Analysis. Department of Computer Science & Engineering, Washington University in St. Louis, 2006.

[6] Benoît Claise. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954. Oct. de 2004. DOI: 10.17487/RFC3954.

[7] P4 Language Consortium. *Behavioral Model v2 (BMv2)*. https://github.com/p4lang/behavioral-model. [Online; accedido el 1-Nov-2025].

[8] P4 Language Consortium. *p4c compiler*. https://github.com/p4lang/p4c. [Online; accedido el 1-Nov-2025].

[9] P4 Language Consortium. *Performance BMv2*. https://github.com/p4lang/behavioral-model/blob/2bdd0b7b2b2ae89faf2720f2158e9842bc6d2dd2/docs/performance.md. [Online; accedido el 2-Nov-2025].

[10] P4 Language Consortium. *Simple Switch*. https://github.com/p4lang/behavioral-model/blob/2bdd0b7b2b2ae89faf2720f2158e9842bc6d2dd2/docs/simple_switch.md. [Online; accedido el 2-Nov-2025].

[11] P4 Language Consortium. *v1model*. https://github.com/p4lang/p4c/blob/a97290474ce3d183b1f6bc4ca4959ebbcdb09b3b/p4include/v1model.p4. [Online; accedido el 1-Nov-2025].

[12] *Data Plane Development Kit*. https://www.dpdk.org/. [Online; accedido el 20-Nov-2025].

[13] *Elastic Stack: (ELK) Elasticsearch, Kibana & Logstash.* https://www.elastic.co/elastic-stack. [Online; accedido el 11-Oct-2025].

[14] *elasticsearch.* https://www.elastic.co/elasticsearch. [Online; accedido el 2-Nov-2025].

[15] Paul Emmerich et al. "MoonGen: A Scriptable High-Speed Packet Generator". En: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, oct. de 2015.

[16] John Emmitt. *RMON: A Closer Look at Remote Network Monitoring.* Kaseya blog. Jun. de 2020. URL: https://www.kaseya.com/blog/rmon-remote-network-monitoring/.

[17] Mark Fedor et al. *Simple Network Management Protocol (SNMP)*. RFC 1157. Mayo de 1990. DOI: 10.17487/RFC1157.

[18] Y Feng. "Multiscale energy network tomography and smartNIC-Accelerated In-band Network telemetry for network internal state monitoring". Tesis doct. Ago. de 2021. DOI: https://catalog.caida.org/paper/2021_y_feng_tamu_1972.

[19] Yixiao Feng et al. "A SmartNIC-Accelerated Monitoring Platform for In-band Network Telemetry". En: *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*. 2020, págs. 1-6. DOI: 10.1109/LANMAN49260.2020.9153279.

[20] Mesh Flinders y Ian Smalley. *What is network traffic analysis?* https://www.ibm.com/think/topics/network-traffic-analysis. [Online; accedido el 26-Oct-2025]. Abr. de 2024.

[21] David Harrington, Bert Wijnen y Randy Presuhn. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*. RFC 3411. Dic. de 2002. DOI: 10.17487/RFC3411.

[22] Frederik Hauser et al. "A survey on data plane programming with P4: Fundamentals, advances, and applied research". En: *Journal of Network and Computer Applications* (2023). DOI: https://doi.org/10.1016/j.jnca.2022.103561.

[23] IBM Corporation. *SNMP commands.* https://www.ibm.com/docs/en/itcam-transactions/7.4.0?topic=scripts-snmp-commands. IBM Tivoli Composite Application Manager for Transactions, Version 7.4.0.0.

[24] *kibana.* https://www.elastic.co/kibana. [Online; accedido el 2-Nov-2025].

[25] Changhoon Kim et al. "In-band Network Telemetry via Programmable Dataplanes". En: 2015. URL: https://api.semanticscholar.org/CorpusID:15782087.

[26] James F. Kurose y Keith W. Ross. "Computer Networks: a top-down approach, 7th edition". En: 2017. URL: https://api.semanticscholar.org/CorpusID:182560699.

[27] *logstash*. https://www.elastic.co/logstash. [Online; accedido el 2-Nov-2025].

[28] *LuaJIT*. https://luajit.org/. [Online; accedido el 20-Nov-2025].

[29] Masoud Moshref et al. "Flow-level State Transition as a New Switch Primitive for SDN". English. En: *Proceedings of the ACM SIGCOMM Conference*. ACM. Chicago, IL, USA, 2014, págs. 61-66.

[30] Namrata, S. Ravi Shankar y Suresh Babu Kandukuri. "Network Management Using SNMP". En: *International Journal of Advanced Research in Engineering and Technology* 10.3 (2019). Available at SSRN: https://ssrn.com/abstract=3527474, págs. 81-86.

[31] *Netronome Network Flow Processor. NFP SDK version 6.4. Network Flow C Compiler User's Guide*. Manual. Netronome.

[32] *Network Flow Processor SDK. NFP SDK 6.1-preview*. Release Notes. Netronome.

[33] P4 Language Consortium. *P4_16 Language Specification, Version 1.2.5*. 11 October 2024. Oct. de 2024. URL: https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.html.

[34] P4 Language Consortium. *P416 Portable Switch Architecture (PSA), Version v1.2, 2025-10-03*. 10 March 2025. Mar. de 2025. URL: https://p4lang.github.io/p4-spec/docs/PSA.pdf.

[35] P4 Language Consortium. *The P4 Language Specification, Version 1.1.0*. 27 January 2016. Ene. de 2016. URL: https://p4.org/wp-content/uploads/sites/53/p4-spec/p4-14/v1.1.0/tex/p4.pdf.

[36] P4.org Applications Working Group. *In-band Network Telemetry (INT) Dataplane Specification, Version 2.1*. Inf. téc. Contributions from Alibaba, Arista, CableLabs, Cisco Systems, Dell, Intel, Marvell, Netronome, and VMware. P4.org, nov. de 2020. URL: https://p4.org/wp-content/uploads/sites/53/p4-spec/docs/INT_v2_1.pdf.

[37] Salvatore Pontarelli et al. "Flowblaze: stateful packet processing in hardware". En: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. NSDI'19. Boston, MA, USA: USENIX Association, 2019, págs. 531-547. ISBN: 9781931971492.

[38] *Programming Netronome Agilio SmartNICs. NFP-4000 and NFP-6000 family: supported programming models*. White paper. Netronome.

[39] *Programming NFP with P4 and C*. White paper. Netronome.

[40] *Redis: The Real-time Data Platform*. https://redis.io/. [Online; accedido el 1-Nov-2025].

[41] *Repositorio del sistema de monitorización INT basado en SmartNICs*. https://gitlab.fing.edu.uy/smartlab/monitorizacion-rc. [Online; accedido el 17-Nov-2025].

[42] *Respuesta de soporte técnico en el grupo Open-NFP "SmartNIC - Throughput degradation on sending packets to the Host"*. https://groups.google.com/g/open-nfp/c/OrXsu6Lgayw/m/rgge6Ku0BAAJ. [accedido el 18-Nov-2025]. 2019.

[43] Matías Richart et al. "LQ-GNN: A Graph Neural Network Model for Response Time Prediction of Microservice-Based Applications in the Computing Continuum". En: *IEEE Transactions on Parallel and Distributed Systems* (2025). DOI: 10.1109/TPDS.2025.3564214.

[44] Anirudh Sivaraman et al. "Packet Transactions: High-Level Programming for Line-Rate Switches". En: *Proceedings of the 2016 ACM SIGCOMM Conference*. Association for Computing Machinery, 2016. DOI: 10.1145/2934872.2934900.

[45] Haoyu Song. "Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane". En: Association for Computing Machinery, 2013. DOI: 10.1145/2491185.2491190.

[46] *The Joy of Micro-C*. https://groups.google.com/group/open-nfp/attach/8031894d1c32d/the-joy-of-micro-c_fcjSfra.pdf. [Online; accedido el 1-Nov-2025].

[47] Steven Waldbusser. *Remote Network Monitoring Management Information Base*. RFC 1271. Nov. de 1991. DOI: 10.17487/RFC1271.

[48] Steven Waldbusser. *Remote Network Monitoring Management Information Base*. RFC 1757. Feb. de 1995. DOI: 10.17487/RFC1757.

[49] Steven Waldbusser. *Remote Network Monitoring Management Information Base*. RFC 2819. Mayo de 2000. DOI: 10.17487/RFC2819.

[50] Nofel Yaseen. "From Counters to Telemetry: A Survey of Programmable Network-Wide Monitoring". En: *Network* (2025). DOI: 10.3390/network5030038.

[51] Bin Zhang. *Traffic data from real network environment (version 1)*. figshare. 2025. DOI: 10.6084/m9.figshare.28380347.v1.

Anexo A

Bloques de código y pseudocódigo

A.1. Pseudocódigo *Micro-C Sink*

```
function pif_plugin_save_in_hash(headers, match_data):

    // 1. Obtener hash (5-tupla del flujo)
    if get_hash_key(headers, hash_key) < 0:
        return FORWARD
    hash_value = hash_me_crc32(hash_key) & (FLOWCACHE_ROWS - 1)
    ring_index = hash_value & (NUM_RINGS - 1)

    // 2. Buscar entrada en la tabla o decidir reemplazo
    semaphore_down(global_semaphores[hash_value])
    entry = buscar_entrada_en_bucket(
               hash_value,
               hash_key,
               out victim_entry,
               out evict_selected
           )

    // 3. Si debe reemplazarse una entrada, exportarla al ring buffer G
    if evict_selected:
        exportar_entry_a_ring_G(victim_entry, ring_index)
        limpiar(victim_entry)
        entry = victim_entry

    // 4. Actualizar o inicializar la entrada del flujo
    actualizar_timestamp(entry)
    incrementar_contador(entry)

    if primera_vez(entry):
```

```
        guardar_clave(entry, hash_key)
        inicializar_timestamps_y_metadatos(entry)

    // 5. Procesar mtricas INT por nodo
    para cada nodo INT del paquete:
        leer mtricas del nodo
        generar T-event si valor supera umbral
        comparar con valor previo y generar C-event si cambia suficiente
        actualizar latest y promedio del nodo
        acumular hop-latency para mtrica extremo-a-extremo

    semaphore_up(global_semaphores[hash_value])

    // 6. Evaluar mtricas E2E
    si e2e_latency supera umbral:
        generar T-event E2E
    si cambio absoluto en e2e_latency supera umbral:
        generar C-event E2E

    return FORWARD
end function
```

A.2. Estructuras principales en memoria

```
#define FLOWCACHE_ROWS (1 << 18)
#define BUCKET_SIZE 12
#define MAX_INT_NODES 5
#define IP_PROTO_UDP 0x11
#define IP_PROTO_TCP 0x6
#define NUM_RINGS 8
#define RING_SIZE (1 << 17)

typedef struct int_metric_sample {
  uint32_t node_id;          /* Node ID */
  uint32_t hop_latency;      /* Hop latency */
  uint32_t queue_occupancy;  /* Queue occupancy */
  uint32_t egress_interface_tx; /* Egress interface transmission */
} int_metric_sample;

typedef struct int_metric_info {
  int_metric_sample latest[MAX_INT_NODES];
  int_metric_sample average[MAX_INT_NODES];
  uint32_t node_count;
} int_metric_info;

typedef struct bucket_entry {
```

```
  uint32_t key[4]; /* ipv4.src_addr, ipv4.dst_addr, (src_port << 16) |
      dst_port, ipv4.protocol */
  uint64_t first_packet_timestamp; /* Timestamp in nanoseconds */
  uint64_t last_update_timestamp; /* Timestamp in nanoseconds */
  int_metric_info int_metric_info_value;
  uint32_t packet_count;
  uint32_t request_meta; // bits 015: request_id, bit 16: is_response,
      bits 1731: reserved
  uint32_t _padding2;
} bucket_entry;

typedef struct bucket_list {
  struct bucket_entry entry[BUCKET_SIZE];
} bucket_list;

__export __emem bucket_list int_flowcache[FLOWCACHE_ROWS];
```

A.3. Pseudocódigo *Host_ Reader*

```
main:

    // 1. Inicializar spoolers
    iniciar_spooler()
    iniciar_event_spooler()

    // 2. Abrir dispositivo NFP y obtener manejador CPP
    h_nfp = abrir_dispositivo()
    h_cpp = obtener_cpp(h_nfp)

    // 3. Obtener smbolos en runtime (direcciones de ring buffers y
        metadatos)
    buscar_simbolos_rings_G()
    buscar_simbolos_rings_I()

    // 4. Crear reas CPP para cada ring (G e I)
    para cada ring en NUM_RINGS:
        asignar_area_ring_G
        asignar_area_ring_I
        asignar_area_meta_G
        asignar_area_meta_I

    // 5. Crear hilos que procesan eventos de los RI
    dividir RI entre cantidad de hilos
    por cada hilo:
         lanzar worker(event_ring_worker)

    // 6. Bucle principal
```

```
mientras no fin:
    para cada RG:
        leer metadatos

        repetir hasta BATCH_SIZE veces:
            si el ring est  vaco  -> salir del batch
            leer entrada del ring
            enviar a spooler

        si se consumi  algo -> actualizar metadatos

// 7. Esperar fin de hilos y liberar recursos
unir_hilos_workers_I()
liberar_areas_CPP()
cerrar_dispositivo()
detener_spoolers()
```

Anexo B

Guía de uso de la plataforma desarrollada

Este anexo presenta una guía práctica para la utilización de la plataforma desarrollada. Se describen los pasos necesarios para compilar y cargar el *INT Sink* en las SmartNICs Netronome, generar y enviar tráfico sintéticamente instrumentado, operar la capa de visualización basada en ELK y emplear el *dissector* de Wireshark incluido en el repositorio.

Estructura del repositorio

El repositorio asociado al proyecto se encuentra en [41] y se organiza de la siguiente forma. Dentro del directorio `src/`:

- `bin/`: scripts ejecutables para facilitar tareas de compilación, carga de programas en SmartNICs, inicialización de servicios, entre otros.
- `dissectors/`: *dissector* de Wireshark para visualizar paquetes INT.
- `elastic/`: configuración de Elasticsearch y Filebeat.
- `evaluation/`: herramientas y scripts utilizados para la evaluación experimental.
- `host/`: programas que se ejecutan en el host para leer los *ring buffers* y exportar metadatos a archivos `ndjson`.
- `sink/`: programa P4 y rutinas en C asociadas al *INT Sink*.
- `traffic_generator/`: generador de tráfico con inserción de metadatos INT.
- `transit_hop/`: programa P4 para la prueba de concepto del *INT Transit Hop* en `bmv2`.

- `utils/`: utilidades varias (ej: recarga del *dissector*).
- `wire/`: programa P4 para pruebas de *baseline*.

Uso del INT Sink

Compilación del programa P4

Para compilar el programa del *Sink*:

```
cd sink
../bin/p4 build
```

Carga del programa en la SmartNIC

Para cargar el programa compilado en la tarjeta:

```
../bin/p4 design-load
```

Lectura de metadatos INT en el host

El host ejecuta un servicio encargado de leer los *ring buffers*:

```
cd ../host
sudo make restart
```

Generación y envío de tráfico INT hacia el Sink

Generación de una traza con INT

Para generar una traza `.pcap` con metadatos INT:

```
cd traffic_generator
```

Opcionalmente, modificar los parámetros de `config.yaml`. Luego ejecutar:

```
python3 int_injector.py
```

Reenvío de la traza hacia el Sink

Usando tcpreplay

```
sudo tcpreplay -i <INTERFAZ> --topspeed generated_int.pcap
```

Usando MoonGen

```
./bin/dpdk_bind_if vf0_0
sudo ~/MoonGen/setup-hugetlbfs.sh
sudo MoonGen evaluation/replay_pcap.lua 0 -n 1 \
     traffic_generator/generated_int.pcap
```

Uso del stack ELK

Inicialización de los servicios

El proyecto incluye scripts para verificar e inicializar Elasticsearch, Kibana y Filebeat:

- `bin/ensure_stack`: verifica que los servicios estén activos y los inicia si es necesario.
- `bin/setup_filebeat`: configura Filebeat para monitorear los archivos `ndjson`.

Acceso a Kibana

Para acceder a Kibana se recomienda establecer un túnel SSH:

```
ssh -J nombre.apellido@lulu.fing.edu.uy \
    -L 5601:localhost:5601 host@ip
```

Luego abrir en el navegador:

```
http://localhost:5601
```

Uso del dissector de Wireshark

Para habilitar o recargar el *dissector*:

```
utils/reload_dissector.sh
```

Luego, en Wireshark:

```
Analyze -> Reload Lua Plugins}
```

Requisitos de software y hardware

La plataforma requiere:

- SmartNICs Netronome Agilio CX 2×10GbE.
- Kernel Linux compatible (ej. Ubuntu 18.04.6, kernel 4.18.0-15-generic).
- SDK de Netronome y controladores NFP.
- Soporte para *hugepages*.
- DPDK y MoonGen (incluyendo LuaJIT).
- Elasticsearch, Kibana y Filebeat.
- Wireshark y tshark.
- Python3 y bibliotecas adicionales (p. ej. `scapy`).