UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Monitorización para el Control de Congestión

Informe de Proyecto de Grado presentado por

Ian Arazny Casanovas, Favio Cardoso y Maria Techera Alberro

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, 5 de mayo de 2026

Agradecimientos

A nuestras familias y amistades, por su apoyo constante a lo largo de este año de trabajo, por la paciencia en los momentos más exigentes y por el acompañamiento que hizo posible culminar este proyecto.

A nuestros tutores, por su dedicación, orientación y disponibilidad permanente ante cada duda o desafío. Sus comentarios y sugerencias fueron fundamentales para el desarrollo y la calidad de este trabajo.

A todas las personas que de una forma u otra aportaron al desarrollo de este proyecto: ¡muchísimas gracias!

In memoriam

En memoria de mi madre, por enseñar amor infinito y brindar un soporte incondicional.
Aunque ya no estés, tu cariño sigue guiándome.

Resumen

El presente proyecto aborda el diseño e implementación de un sistema de telemetría inteligente para redes de alta capacidad, con el objetivo de anticipar condiciones de congestión y degradación del rendimiento a partir de datos recolectados en tiempo real. La propuesta combina mecanismos de inspección de bajo nivel, ejecutadas directamente en la tarjeta de red, con modelos de aprendizaje automático supervisado para el análisis predictivo del tráfico.

El sistema se desarrolló sobre una arquitectura jerárquica en tres niveles: la tarjeta de red programable (*Smart Network Interface Card (SmartNIC)*), el núcleo del sistema operativo Linux, y el espacio de usuario. En la SmartNIC se ejecutaron programas basados en la tecnología *extended Berkeley Packet Filter (eBPF)* y su extensión de procesamiento en red *eXpress Data Path (XDP)*, capaces de capturar y clasificar paquetes directamente en el plano de datos. Estos programas operan en modo de descarga parcial (*offload*), de modo que parte del procesamiento se realiza en el hardware de la tarjeta, reduciendo la carga sobre el procesador principal.

En el nivel del núcleo se implementó la agregación de las métricas recolectadas en la SmartNIC, mientras que en el espacio de usuario se desarrollaron módulos en lenguaje C encargados de leer los datos, almacenarlos y exponerlos para su análisis. La persistencia se realizó mediante la base de datos TimescaleDB, una extensión de PostgreSQL optimizada para series temporales, y la visualización, con un programa de consola.

El conjunto de datos obtenido combina métricas de red y del sistema operativo, capturadas mediante telemetría en SmartNIC fueron utilizadas para entrenar modelos supervisados de clasificación y regresión basados en el algoritmo *Extreme Gradient Boosting (XGBoost)*. Este enfoque se seleccionó por su robustez frente a ruido, su capacidad para modelar relaciones no lineales y su eficacia demostrada en problemas de análisis de tráfico de red. El modelo clasificador mostró un desempeño estable y confiable en la detección de eventos de congestión, mientras que el modelo regresor permitió analizar la evolución temporal del tráfico, aunque con menor capacidad de generalización debido a la variabilidad y naturaleza dinámica de la red. En conjunto, los resultados confirman la viabilidad de aplicar técnicas de aprendizaje supervisado en entornos de monitoreo inteligente, resaltando la necesidad de enfoques adaptativos que acompañen la evolución del tráfico en redes de alta capacidad.

Palabras clave: *eBPF, SmartNIC, predicción de congestión, monitorización en tiempo real, Congestion Control.*

Índice general

Lista de acrónimos

Capítulo 1

Introducción

El crecimiento exponencial del tráfico en redes de alta capacidad, impulsado por servicios en la nube, centros de datos y el Internet of Things (IoT), ha incrementado la necesidad de mecanismos de monitorización más inteligentes y eficientes (Guo y cols., 2015). Sin embargo, las técnicas tradicionales de telemetría basadas en muestreo o en contadores como Simple Network Management Protocol (SNMP) (Harrington, Wijnen, y Presuhn, 2002), NetFlow (Claise, 2004) o sampled Flow (sFlow) (Panchen, McKee, y Phaal, 2001a) resultan insuficientes para capturar la complejidad y el dinamismo del tráfico moderno (Miano, Lettieri, Antichi, y Procissi, 2024; Boutaba y cols., 2018). La detección temprana de congestión, anomalías y patrones de rendimiento exige capacidades de procesamiento en tiempo real y una mayor proximidad al plano de datos (I. Corporation, 2020; Zhou y cols., 2020).

En este contexto, la programabilidad de red mediante tecnologías como *extended Berkeley Packet Filter/eXpress Data Path (eBPF/XDP)* y *Programming Protocol-independent Packet Processors (P4)* ha permitido trasladar tareas de análisis y control directamente al plano de datos, reduciendo la latencia y la dependencia del Central Processing Unit (CPU) principal (Bosshart y cols., 2014). Incorporado al kernel[1] de Linux desde la versión 3.18, extended Berkeley Packet Filter (eBPF) posibilita la ejecución de programas seguros y aislados dentro del kernel, permitiendo acceder a estructuras internas sin modificar su código fuente y habilitando un procesamiento altamente eficiente (Gregg, 2019).

Si bien P4 ofrece una gran expresividad para definir el procesamiento de paquetes de forma independiente del hardware, su adopción práctica se encuentra limitada por la disponibilidad de *targets* compatibles y por la complejidad de los entornos de despliegue (Gupta y cols., 2018; Silicon Valley Business Journal, 2023). En contraste, eBPF se integra nativamente al kernel de Linux, lo que facilita el desarrollo, depuración y despliegue en sistemas operativos comunes, además de contar con soporte directo en *SmartNICs* como la Netronome Agilio

[1] Es el componente central de un sistema operativo que actúa como un puente esencial entre el hardware y el software.

CX[2] (Netronome Systems, Inc., 2019). Por esta razón, y siguiendo la consigna original del proyecto, se optó por un enfoque basado en eBPF/XDP, que equilibra flexibilidad, compatibilidad y rendimiento.

La elección de las tecnologías utilizadas responde tanto a criterios técnicos como a los lineamientos establecidos en la consigna del proyecto. Se optó por el uso de eBPF y su extensión XDP en lugar del lenguaje *P4*, dado que la propuesta debía desarrollarse sobre el ecosistema Linux y ejecutarse en una SmartNIC Netronome Agilio CX, plenamente compatible con eBPF en modo *offload*. Esta decisión permitió aprovechar la integración nativa de eBPF con el kernel para realizar procesamiento de paquetes en el plano de datos, reduciendo la carga del CPU, al tiempo que se combinaron las ventajas del análisis predictivo mediante modelos de aprendizaje automático. Además, se destaca la relevancia actual de eBPF, ampliamente adoptado en entornos de *cloud computing*[3] para observabilidad, seguridad, trazado de rendimiento y control de políticas en tiempo real, lo que refuerza su valor como tecnología transversal en redes modernas.

Más allá de su aplicación en telemetría, eBPF se ha consolidado como una tecnología transversal en el ecosistema de sistemas y redes modernas(IO Visor Project, 2023). Actualmente, es utilizada ampliamente en entornos de *cloud computing* para tareas de observabilidad avanzada, detección de intrusiones, trazado de rendimiento y control dinámico de políticas (Sharma y Nadig, 2024). Estas capacidades evidencian su versatilidad para operar desde el plano de datos hasta la capa de aplicación sin comprometer la estabilidad del sistema.

De forma complementaria, las *SmartNICs* incorporan procesadores dedicados capaces de ejecutar programas eBPF en paralelo al sistema (Høiland-Jørgensen y cols., 2018). Esto posibilita el *offload*[4] de funciones de filtrado, clasificación o recolección de métricas directamente sobre el tráfico entrante (Miano y cols., 2024), mejorando la eficiencia y permitiendo análisis en tiempo casi real. Asimismo, tecnologías como *Data Plane Development Kit (DPDK)*[5] amplían las capacidades del plano de datos al habilitar procesamiento de tráfico de alta velocidad en el espacio de usuario(DPDK Project, 2018).

Paralelamente, el *aprendizaje automático* ha emergido como una herramienta clave para el análisis y la predicción de fenómenos de red. Los modelos supervisados permiten identificar dependencias no lineales entre métricas de tráfico y anticipar condiciones de congestión o degradación del servicio. Boutaba et al. (Boutaba y cols., 2018) destacan que los enfoques basados en datos han transformado la gestión de redes al posibilitar la correlación entre múltiples ni-

[2] Las SmartNICs de la serie Agilio CX de Netronome incorporan soporte para la ejecución y descarga de programas eBPF directamente en el hardware de red, lo que permite acelerar tareas de filtrado, conteo y análisis de tráfico en el plano de datos

[3] Es la entrega de servicios informáticos (como servidores, almacenamiento, bases de datos, redes, software y análisis) a través de Internet. En lugar de poseer y mantener su propia infraestructura física, las empresas y los usuarios pueden acceder a estos recursos bajo demanda, pagando solo por lo que usan.

[4] Del inglés *to offload*: descargar o delegar tareas de procesamiento desde la CPU principal hacia otro componente, como una SmartNIC o dispositivo acelerador.

[5] DPDK es un conjunto de bibliotecas y controladores que permite procesar paquetes a alta velocidad en espacio de usuario, evitando las interrupciones y el overhead del kernel.

veles de telemetría y el ajuste dinámico de recursos, lo cual se complementa con propuestas recientes que aplican analítica avanzada sobre flujos y métricas recolectadas en el plano de datos (Miano y cols., 2024).

En particular, algoritmos como *Extreme Gradient Boosting (XGBoost)*[6] (Chen y Guestrin, 2016) y *Random Forest (RF)* [7] (Breiman, 2001) han mostrado resultados prometedores en la predicción de métricas críticas —como congestión, latencia o tamaño de cola—(Zhang, Patras, y Haddadi, 2019) en redes de centros de datos.

1.1. Definición del Problema y Objetivos

El problema central que motiva este trabajo radica en la dificultad de procesar métricas de tráfico en tiempo real dentro de redes de alta capacidad, considerando la velocidad de generación de datos, que en datacenters puede alcanzar millones de flujos por segundo(Liu, Gao, Liu, Zhang, y Foh, 2017), las limitaciones de los enfoques basados únicamente en CPU de propósito general y las restricciones de hardware en las SmartNICs, que imponen límites tanto de memoria como de tamaño de programa.

En consecuencia, surge la necesidad de explorar arquitecturas híbridas que combinen la capacidad de procesamiento paralelo del hardware programable con la flexibilidad del software, incorporando además técnicas de aprendizaje automático que permitan anticipar condiciones críticas del tráfico.

El objetivo general del proyecto es diseñar y evaluar una arquitectura de monitorización inteligente basada en telemetría programable[8] con eBPF/XDP y modelos de aprendizaje automático supervisado, capaz de recolectar métricas en tiempo real desde SmartNICs y predecir condiciones de congestión.

Para alcanzar este propósito, se plantean tres líneas de trabajo:

1. Replicar la arquitectura híbrida propuesta en (Miano y cols., 2024), que ejecuta funciones de recolección y clasificación de tráfico integrando *offload*[9] parcial de operadores a una SmartNIC Netronome Agilio CX NFP-4000.

[6] Algoritmo de gradient boosting altamente optimizado que mejora la eficiencia y generalización de los modelos basados en árboles, ampliamente usado en problemas de clasificación y regresión.

[7] Método de ensamblado propuesto por Breiman que combina múltiples árboles de decisión entrenados sobre subconjuntos aleatorios de datos y atributos, mejorando la estabilidad y precisión del modelo .

[8] La **telemetría programable** permite configurar la red para recolectar datos directamente desde el plano de datos, configurando qué métricas se miden, dónde y con qué frecuencia, sin depender de mecanismos de sondeo o exportación periódica. A diferencia de los enfoques tradicionales como SNMP o NetFlow, habilita una observación continua y granular mediante hardware y software programables (*e.g.*, eBPF, P4 o SmartNICs).

[9] El modo *offload* permite ejecutar programas eBPF directamente en el hardware de la tarjeta de red (SmartNIC), transfiriendo la lógica de procesamiento desde el CPU principal hacia los procesadores integrados del dispositivo. Este enfoque descarga de trabajo al sistema operativo, siendo ideal para tareas de clasificación y telemetría en tiempo real.

2. Incorporar un modelo de aprendizaje automático supervisado para la predicción de métricas de red y congestión a partir de las métricas recolectadas por el sistema de telemetría. Adicionalmente, desarrollar dos modelos de aprendizaje automático supervisado usando XGBoost entrenado con métricas de flujo para predecir congestión mediante clasificación y predecir métricas de red mediante regresión.

3. Evaluar experimentalmente el desempeño global del sistema en términos de utilización de recursos —CPU, memoria y throughput— bajo diferentes modos de ejecución de los programas eBPF, comparando la operación en modo *offload* en la SmartNIC con la ejecución en modo *driver* dentro del kernel.

El alcance del proyecto comprende:

- La implementación de programas eBPF en modo offload sobre una SmartNIC Netronome Agilio CX, diseñados para realizar tareas de clasificación de paquetes, conteo de flujos y recolección de métricas de tráfico.
- El desarrollo de aplicaciones de análisis de tráfico tales como la medición de flujos Transmission Control Protocol (TCP) y la caracterización de eventos relacionados con congestión.
- La integración de las métricas recolectadas con modelos de aprendizaje automático supervisado, capaces de predecir condiciones asociadas al estado de la red.
- La evaluación comparativa del rendimiento entre la ejecución en software (en CPU host) y el enfoque con offload en la SmartNIC, considerando métricas como packets per second, throughput, uso de CPU y precisión de las predicciones.

Quedan fuera del alcance aspectos como la integración con arquitecturas de telemetría distribuidas basadas en In-band Network Telemetry (INT), la extensión a entornos de producción a gran escala o el uso de técnicas avanzadas de aprendizaje profundo. Estas líneas se plantean como posibles trabajos futuros.

El entorno de laboratorio utiliza tráfico sintético generado mediante `iperf3` (ESnet, 2019) y *Hierarchical Token Bucket (HTB) + Fair Queuing Controlled Delay (fq_codel) con opción Explicit Congestion Notification (ECN)* (Nichols y Jacobson, 2018; Ramakrishnan, Floyd, y Black, 2001). No se abordan escenarios de producción.

Este trabajo se inspira en la arquitectura planteada en (Miano y cols., 2024), donde se combinan programas eBPF/XDP ejecutados en modo offload sobre la SmartNIC con un pipeline de análisis en espacio de usuario. En este enfoque, parte del procesamiento de paquetes se delega al hardware programable, reduciendo la carga del CPU central y habilitando una recolección de métricas más cercana al plano de datos.

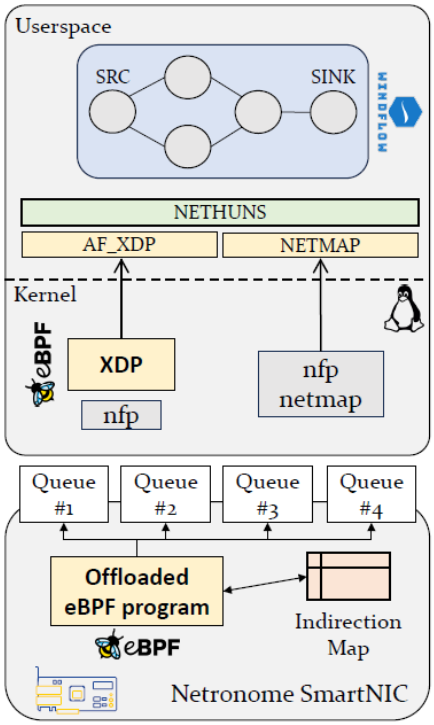

Figura 1.1: Arquitectura de referencia que combina programas eBPF/XDP descargados en la SmartNIC con procesamiento en espacio de usuario (Miano y cols., 2024).

La Figura 1.1 representa la arquitectura utilizada como base. En la parte inferior se encuentra la SmartNIC Netronome Agilio CX, donde se ejecutan en modo *offload* los programas eBPF/XDP. Estos programas procesan paquetes directamente en las colas de recepción (Queue #1–#4), permitiendo realizar tareas de filtrado, clasificación y recolección de métricas sin intervención del CPU principal. El procesamiento se apoya en un *Indirection Map*[10], que distribuye el tráfico entre las diferentes colas de hardware.

En el kernel del host, el controlador `Network Flow Processor (NFP)` expone interfaces compatibles con XDP, Address Family XDP (sockets) (AF_XDP) y *netmap* (Rizzo, 2012), que permiten transferir los paquetes procesados hacia el espacio de usuario con un coste mínimo. En esta capa se ejecuta *Nethuns*, una librería de captura de alto rendimiento que facilita la comunicación entre el plano de datos y las aplicaciones de usuario.

Finalmente, en el espacio de usuario se implementa el pipeline de análisis,

[10]El *Indirection Map* es una estructura de mapeo utilizada por los sistemas de multicolas (RSS — Receive Side Scaling) para distribuir paquetes entrantes entre distintas colas de recepción.

compuesto por módulos `SRC` y `SINK` para la ingestión y procesado de flujos. Este pipeline utiliza las métricas recolectadas por los programas descargados en la SmartNIC, y constituye el punto donde se integran los modelos de aprendizaje automático entrenados en este proyecto. La combinación de procesamiento en hardware y análisis avanzado en software permite reducir la carga del host y habilitar capacidades de monitorización en tiempo casi real.

La elección de las tecnologías utilizadas responde tanto a criterios técnicos como a los lineamientos establecidos en la consigna del proyecto. Se optó por eBPF y su extensión XDP en lugar del lenguaje *P4*, dado que la propuesta debía desarrollarse sobre el ecosistema Linux y ejecutarse en una SmartNIC Netronome Agilio CX, plenamente compatible con eBPF en modo *offload*. Esta decisión permitió aprovechar la integración nativa de eBPF con el kernel para realizar procesamiento de paquetes en el plano de datos, reduciendo la carga del CPU, al tiempo que se incorporaron las ventajas del análisis predictivo mediante modelos de aprendizaje automático. Además, eBPF goza de una amplia adopción en entornos de *cloud computing*, donde se utiliza para observabilidad, seguridad y control dinámico de políticas en tiempo real, lo que refuerza su relevancia como tecnología transversal en redes modernas. A diferencia del enfoque original basado en el motor de procesamiento de flujos WindFlow (Mencagli y cols., 2021), en este proyecto se implementaron manualmente los programas eBPF necesarios para la clasificación y el conteo de paquetes, así como para la recolección de métricas de tráfico. Estos programas fueron descargados a la SmartNIC, permitiendo que el procesamiento ocurra en el plano de datos y optimizando el uso de recursos del host.

Los resultados esperados de este proyecto incluyen, además de los objetivos de implementación, la validación experimental de los siguientes aspectos medibles:

- Cuantificar el incremento de throughput obtenido al ejecutar programas eBPF/XDP en modo offload comparado con su ejecución en modo driver.
- Medir la reducción del consumo de CPU y memoria en el host.
- Recolectar métricas precisas de flujos TCP como insumo para modelos predictivos.
- Evaluar la capacidad predictiva del modelo XGBoost para anticipar eventos de congestión.
- Validar experimentalmente la viabilidad del enfoque híbrido telemetría programable + ML supervisado.

Conclusiones preliminares

Los avances obtenidos permiten concluir que la combinación de telemetría programable, SmartNICs y aprendizaje automático supervisado constituye una vía prometedora para la monitorización en datacenters y redes de alta capacidad.

Esta integración permite mejorar el procesamiento de tráfico en tiempo real y anticipar problemas antes de que afecten a los usuarios.

En conjunto, el sistema desarrollado demuestra el potencial de unir telemetría programable y análisis predictivo dentro del plano de datos, sentando las bases para futuras arquitecturas autónomas e inteligentes.

1.2. Contribuciones del trabajo

Este proyecto realiza aportes concretos en el ámbito de la telemetría programable y el aprendizaje automático aplicados a redes de alta capacidad:

1. Implementación de programas eBPF/XDP en modo offload sobre una SmartNIC Netronome Agilio CX.
2. Diseño e integración de una arquitectura completa de monitoreo y predicción, con persistencia en TimescaleDB y visualización en Grafana.
3. Creación de un dataset propio con etiquetado de congestión explícita (Explicit Congestion Notification/Congestion Experienced (ECN-CE)).
4. Entrenamiento y validación de modelos XGBoost sobre tráfico real recolectado.
5. Evaluación comparativa entre ejecución en host y SmartNIC, con evidencia cuantitativa de mejora de rendimiento.
6. Contribución metodológica al demostrar la viabilidad de integrar telemetría programable con analítica predictiva en tiempo real.

Finalmente, el sistema implementado, los scripts de recolección de métricas y los modelos entrenados se publican en el repositorio institucional de la Facultad de Ingeniería, junto con la documentación necesaria para su instalación y despliegue.[11]

[11] Repositorio institucional: https://gitlab.fing.edu.uy/smartlab/monitorizacion-cc

Capítulo 2

Revisión de antecedentes

La monitorización de redes ha sido, desde los orígenes de Internet, un área crítica para garantizar la estabilidad, el rendimiento y la seguridad de las comunicaciones digitales (Svoboda, Ghafir, y Prenosil, 2015). A medida que las redes han crecido en escala y complejidad, también lo han hecho las herramientas necesarias para recolectar, analizar y actuar sobre los datos generados por los dispositivos y flujos de tráfico. Este proceso de evolución ha llevado desde mecanismos simples de sondeo, como el *SNMP*, hasta tecnologías modernas de telemetría programable soportadas en dispositivos especializados como las *SmartNICs*, en combinación con algoritmos avanzados de aprendizaje automático (Yaseen, 2025; Elizalde y cols., 2025).

En una primera etapa, la telemetría de red se sustentó en protocolos como SNMP (Fedor, Schoffstall, Davin, y Case, 1990), NetFlow (Claise, 2004) y sFlow (Panchen y cols., 2001a), ampliamente adoptados en la industria por su simplicidad y bajo costo de implementación. Estos mecanismos permitieron a los administradores obtener métricas de tráfico y estado de los dispositivos de manera periódica (I. Corporation, 2020; Gupta y cols., 2018). Sin embargo, las limitaciones inherentes a este enfoque, tales como la baja granularidad, la latencia en la recolección de datos y la dependencia de procesos centralizados de sondeo, hicieron evidente la necesidad de nuevas soluciones en contextos de redes de gran escala y entornos con requerimientos de tiempo real (Yaseen, 2025).

El advenimiento de entornos caracterizados por *big data*, aplicaciones sensibles a la latencia y servicios distribuidos en la nube impulsó el desarrollo de técnicas de telemetría programable (Aramide, 2024; Brandino y Grampín, 2024). Estas soluciones, basadas en tecnologías como P4, eBPF e INT, habilitaron una instrumentación más flexible y dinámica, permitiendo observar el comportamiento de la red con un nivel de detalle inédito y directamente en el plano de datos (Miano y cols., 2024). Con ello, la monitorización dejó de ser un proceso meramente reactivo y pasó a convertirse en un elemento central en la gestión proactiva de recursos y en la toma de decisiones.

En paralelo, la aparición de las *SmartNICs* introdujo un nuevo paradigma en la arquitectura de redes. Estas tarjetas de interfaz de red inteligentes per-

miten descargar funciones de procesamiento del CPU central, habilitando que la telemetría y otras funciones de red se ejecuten directamente en el dispositivo. Esto no solo mejora la eficiencia, sino que abre la puerta a la integración de técnicas de analítica avanzada y aprendizaje automático supervisado en el propio flujo de datos (Embedded.com, 2023; Infinite Networks, 2023). La sinergia entre telemetría programable, hardware especializado y modelos de *Machine Learning (ML)* ha demostrado ser un enfoque prometedor para anticipar estados de congestión, detectar anomalías y optimizar el rendimiento de redes de alta capacidad (Jain y cols., 2022; Kapoor, Anastasiu, y Choi, 2025).

Este capítulo tiene por objetivo presentar una revisión exhaustiva de los antecedentes relevantes para el presente proyecto. Se comenzará con una descripción de la evolución de las técnicas de telemetría de red, destacando el paso de soluciones tradicionales a arquitecturas programables. Posteriormente, se analizará el rol de las *SmartNICs* en la monitorización de redes, así como la integración del aprendizaje automático supervisado en este campo. Finalmente, se discutirán los principales desafíos, limitaciones y tendencias futuras, estableciendo el marco conceptual y tecnológico sobre el cual se sustenta este trabajo.

2.1. Protocolos tradicionales de telemetría

La monitorización de redes surgió con protocolos diseñados para recolectar información sobre el estado y tráfico de los dispositivos, entre los cuales destacan tres mecanismos fundamentales: el *SNMP*, *NetFlow* y *sFlow*. Estos métodos tradicionales sentaron las bases de la gestión de red, aunque presentan limitaciones frente a las demandas actuales de granularidad, escalabilidad y baja latencia.

Definido en el RFC 1157 (Fedor y cols., 1990), SNMP introdujo un modelo basado en la comunicación entre un gestor central y agentes que exponen variables en una *Management Information Base (MIB)*. Su simplicidad y estandarización facilitaron la administración unificada, pero su dependencia de ciclos de sondeo, baja granularidad y escasa escalabilidad dificultan su aplicación en redes de gran tamaño o alta velocidad.

Además del mecanismo básico de sondeo definido por SNMP, el Internet Engineering Task Force (IETF) desarrolló la familia de módulos *Remote Monitoring (RMON)* con el objetivo de ampliar la capacidad de observación de la red mediante MIBs especializadas. El RFC 3577 (Cole, Romascanu, Kalbfleisch, y Waldbusser, 2003) introduce la arquitectura general de estos módulos, que incorporan contadores de tráfico, estadísticas históricas y detección de eventos sin depender exclusivamente del gestor central. Si bien RMON representó un avance importante al descentralizar parte de la recolección de métricas, su funcionamiento continúa basado en el modelo MIB y en operaciones de consulta periódica, lo que limita su aplicabilidad en redes de alta velocidad donde se requiere granularidad por paquete y respuestas en tiempo real.

Cisco[1] desarrolló NetFlow en los años noventa como una evolución hacia el

[1] **Cisco Systems, Inc.** es una empresa estadounidense líder en redes, telecomunicaciones y soluciones de infraestructura. Publica regularmente el *Cisco Annual Internet Report* (antes

modelo de observación por flujos, definidos por combinaciones de direcciones Internet Protocol (IP), puertos y protocolo (Claise, 2004). A partir de su versión 9, NetFlow se volvió extensible y sirvió de base para el estándar (Aitken, Claise, y Trammell, 2013). Aunque proporciona gran visibilidad y es útil en aplicaciones de planificación, seguridad y facturación, su procesamiento intensivo y la exportación periódica de registros introducen sobrecarga y latencia analítica.

Por su parte, sFlow (Panchen, McKee, y Phaal, 2001b) adoptó un enfoque estadístico mediante el muestreo aleatorio de paquetes, reduciendo significativamente la carga de procesamiento. Un agente sFlow selecciona muestras que un colector analiza para estimar el tráfico total. Su principal fortaleza es la escalabilidad, ya que puede aplicarse en enlaces de alta velocidad sin afectar el rendimiento. No obstante, su precisión se ve limitada en flujos pequeños o eventos breves, lo que reduce su eficacia en detección de anomalías o control de calidad de servicio.

A pesar de sus diferencias, SNMP, NetFlow y sFlow comparten limitaciones estructurales frente a las redes modernas:

- **Escalabilidad:** dificultades para operar sobre volúmenes masivos sin degradar el rendimiento.
- **Granularidad:** incapacidad de recolectar métricas por paquete o en tiempo real.
- **Latencia:** dependencia de ciclos de sondeo o exportación periódica.
- **Reactividad:** diagnóstico post-evento en lugar de anticipación proactiva.

Tabla 2.1: Comparación entre SNMP, NetFlow y sFlow.

Protocolo	**Mecanismo principal**	**Ventajas**	**Limitaciones**
SNMP	Sondeo periódico de variables (MIB)	Sencillez, estandarización, soporte amplio	Baja granularidad y sobrecarga en redes grandes
NetFlow	Exportación de registros de flujo	Visibilidad detallada, múltiples aplicaciones	Alta carga y retraso analítico, requiere colectores
sFlow	Muestreo aleatorio de paquetes + contadores	Escalabilidad y bajo consumo de recursos	Precisión limitada, no detecta eventos transitorios

Visual Networking Index), una referencia mundial en proyecciones de tráfico y tendencias de crecimiento en Internet. Sitio web: https://www.cisco.com/

Estas limitaciones impulsaron la aparición de la *telemetría programable*, que permite instrumentar la red directamente en el plano de datos para obtener visibilidad detallada y en tiempo real.

En este contexto, INT constituye una de las evoluciones más relevantes en telemetría programable. Esta técnica inserta metadatos directamente en los paquetes mientras atraviesan la red, de modo que cada switch o dispositivo participante agrega información sobre su estado local que luego es recolectada en el destino. Al operar a nivel de paquete, INT proporciona una visibilidad precisa y en tiempo real del comportamiento de la red, superando las limitaciones de granularidad y latencia de los mecanismos tradicionales basados en sondeo o muestreo. Si bien su implementación se asocia principalmente a entornos P4, existen propuestas que integran ideas de INT con arquitecturas basadas en eBPF y *SmartNICs*, ampliando su aplicabilidad en plataformas modernas (Xiong y cols., 2024).

La telemetría programable representa así un cambio de paradigma frente a SNMP, NetFlow y sFlow. Al habilitar la observación directa y flexible del tráfico en tiempo real, se convierte en un pilar fundamental para redes de alta capacidad, donde la latencia de monitoreo y la granularidad de datos resultan determinantes. Su integración con *SmartNICs* potencia estas capacidades al acercar el procesamiento a la fuente de datos, reduciendo la carga en los servidores centrales y facilitando la incorporación de algoritmos de aprendizaje automático capaces de anticipar condiciones críticas y optimizar el rendimiento global de la infraestructura.

2.2. Telemetría programable

El crecimiento de las redes modernas en escala, velocidad y complejidad ha puesto en evidencia las limitaciones de los protocolos tradicionales de monitorización. Ante la necesidad de recolectar métricas más precisas, en tiempo real y con un nivel de detalle mayor, surge el paradigma de la telemetría programable (Tan y cols., 2021). Este enfoque permite que los propios dispositivos de red obtengan y reporten métricas en tiempo real desde el plano de datos, habilitando a los dispositivos a generar información de telemetría sin depender exclusivamente de sondeos o exportaciones periódicas (Miano y cols., 2024).

A diferencia de los mecanismos clásicos, la telemetría programable se basa en dispositivos y programas que pueden adaptarse dinámicamente a las necesidades del operador(Gupta y cols., 2018). Esto habilita el despliegue de nuevas métricas sin modificar el hardware subyacente, favoreciendo la flexibilidad y la evolución de la red. Dentro de este paradigma se destacan tres tecnologías fundamentales: P4, eBPF/XDP e INT, que habilitan una instrumentación flexible y de bajo costo computacional (Gregg, 2019).

2.2.1. P4: un lenguaje para el plano de datos

P4 es un lenguaje de programación diseñado específicamente para describir el procesamiento de paquetes en el plano de datos de dispositivos de red. Su principal ventaja radica en la independencia de hardware: un programa escrito en P4 puede compilarse para distintos targets, como ASICs, FPGAs o SmartNICs, siempre que el dispositivo soporte el modelo de arquitectura definido.

En el contexto de telemetría, permite definir reglas de inserción de metadatos en los paquetes, recolectando información como latencia, saltos de red o estado de colas. Esto posibilita la implementación de esquemas de INT, donde los propios paquetes transportan la información de monitoreo mientras atraviesan la red (Kfoury y cols., 2024). Este modelo elimina la necesidad de sondas externas y ofrece visibilidad detallada del tráfico con mínima latencia.

2.2.2. eBPF: instrumentación flexible en sistemas Linux

eBPF es una tecnología que extiende el kernel de Linux permitiendo ejecutar programas verificados en un entorno seguro y de alto rendimiento. Nació como una mejora del clásico Berkeley Packet Filter (McCanne y Jacobson, 1993), pero ha evolucionado hacia una plataforma general de instrumentación y computación dentro del kernel, capaz de actuar en distintas capas del sistema operativo. Su funcionamiento se basa en la carga de pequeños programas compilados a un bytecode[2] intermedio, que son validados por un verificador de seguridad y luego traducidos mediante compilación *Just-In-Time (JIT)*[3] a código nativo específico del procesador. Este proceso garantiza que los programas no accedan a memoria fuera de límites ni generen bucles infinitos, preservando la estabilidad del kernel (Gregg, 2019).

Cada programa eBPF se asocia a un contexto determinado (red, sistema, trazas, etc.) y se ejecuta de forma aislada dentro de una máquina virtual embebida en el kernel. Este diseño permite extender funcionalidades sin modificar el código fuente del sistema operativo, manteniendo la seguridad mediante la verificación estática del bytecode. El intercambio de datos con el espacio de usuario se realiza a través de estructuras llamadas *mapas eBPF*, que funcionan como canales de comunicación compartidos y persistentes entre el kernel y las aplicaciones externas. Estos mapas son la base para construir pipelines de observabilidad en tiempo real integrados con sistemas como TimescaleDB o Grafana[4].

[2] Es un código intermedio, compilado a partir de un código fuente de alto nivel, diseñado para ser ejecutado por una máquina virtual, no directamente por un procesador.

[3] La compilación *JIT* es una técnica en la que el código intermedio se traduce a código máquina en el momento de ejecución, en lugar de hacerlo previamente. Esto permite adaptar la generación de código al procesador real y mejorar el rendimiento sin necesidad de recompilar el sistema completo.

[4] **Grafana** es una plataforma de visualización y análisis de datos que permite crear paneles dinámicos para métricas en tiempo real. Se integra con diversas fuentes de datos, como Prometheus, InfluxDB o TimescaleDB, y es ampliamente utilizada en entornos de observabilidad y monitoreo de infraestructura.

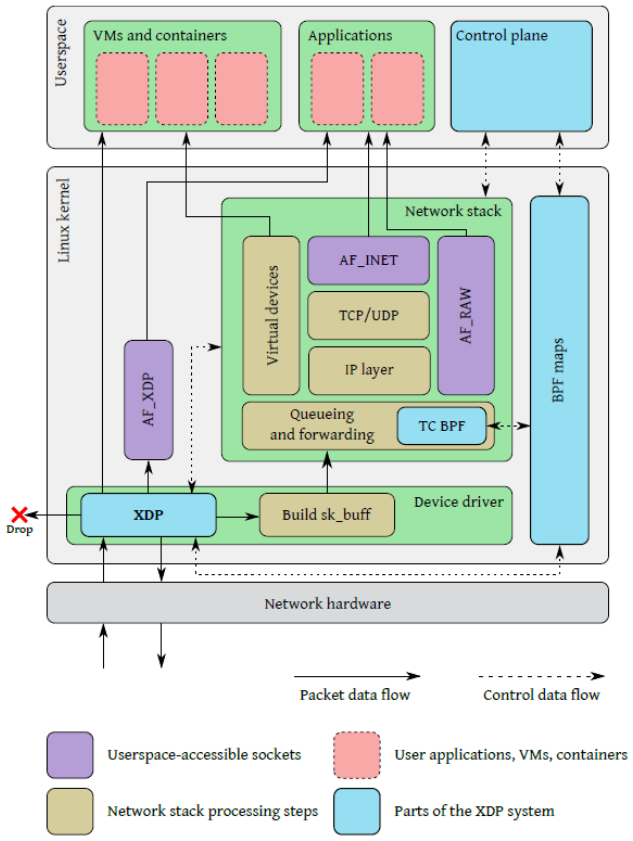

Figura 2.1: Integración de XDP con la pila de red de Linux. Al recibir un paquete, antes de procesar sus datos, el controlador del dispositivo ejecuta un programa eBPF en el hook principal de XDP. Este programa puede descartar paquetes, reenviarlos por la misma interfaz de recepción, redirigirlos a otra interfaz o al espacio de usuario mediante sockets AF_XDP especiales, o permitirles llegar a la pila de red habitual, donde un hook TC BPF independiente puede realizar un procesamiento adicional antes de que los paquetes se pongan en cola para su transmisión. Los distintos programas eBPF pueden comunicarse entre sí y con el espacio de usuario mediante mapas BPF. Para simplificar el diagrama, solo se muestra la ruta de entrada. (Høiland-Jørgensen y cols., 2018).

En entornos de red, eBPF habilita tareas como clasificación de paquetes, conteo de flujos, medición de latencia y extracción de métricas directamente en el plano de datos. Su arquitectura está estrechamente vinculada a la *XDP*, un mecanismo que permite ejecutar programas eBPF en la fase más temprana del pipeline de recepción de paquetes, incluso antes de que estos ingresen a la pila Transmission Control Protocol/Internet Protocol (TCP/IP). Esto reduce

drásticamente el overhead de procesamiento y habilita un rendimiento comparable al de soluciones basadas en hardware especializado. De acuerdo con Høiland-Jørgensen et al. (Høiland-Jørgensen y cols., 2018), XDP mantiene tasas de procesamiento de línea de 10 – 40 Gbps sin sacrificar la flexibilidad de programación, lo que lo posiciona como un punto intermedio entre el kernel tradicional y el hardware de red programable.

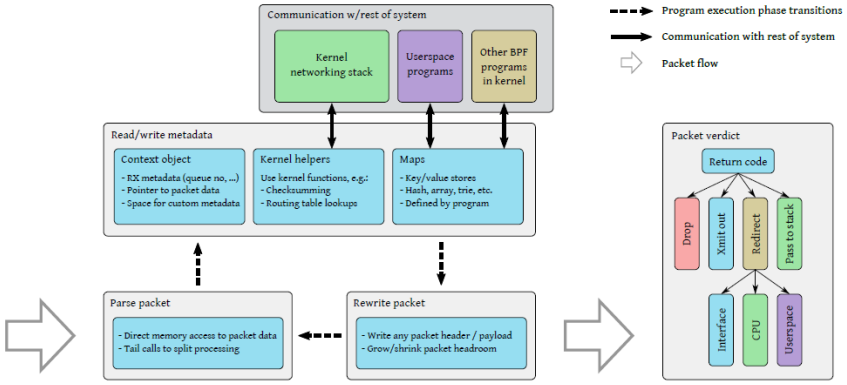

Figura 2.2: Flujo de ejecución de un programa XDP típico. El programa analiza el paquete, accede a metadatos y mapas del kernel, y finalmente emite un veredicto (por ejemplo, `DROP`, `REDIRECT` o `PASS`). Adaptado de Høiland-Jørgensen et al.(Høiland-Jørgensen y cols., 2018).

Una de las características más destacadas de eBPF es su extensibilidad. A través de colecciones de bibliotecas y frameworks como *bcc*[5], *libbpf*[6], *bpftrace*[7] y *CO-RE* (Compile Once, Run Everywhere), se ha consolidado un ecosistema maduro que simplifica la creación de herramientas de observabilidad, seguridad y redes. Este ecosistema ha permitido el surgimiento de proyectos de código abierto —como Cilium[8], Falco[9], Hubble[10] o Katran[11]— que aprovechan eBPF

[5]BPF Compiler Collection (BCC): Es un conjunto de herramientas para crear programas de rastreo y manipulación del kernel de Linux

[6]Es una biblioteca de C (`https://en.wikipedia.org/wiki/C_(programming_language)`) para trabajar con programas eBPF en el kernel de Linux. `https://github.com/libbpf/libbpf`

[7]Es un lenguaje de seguimiento de alto nivel para Linux que utiliza el subsistema eBPF para obtener información sobre el rendimiento del sistema en tiempo real

[8]**Cilium** es una plataforma de conectividad y seguridad para contenedores basada en eBPF, que provee filtrado de nivel 7, políticas de red y observabilidad avanzada. Sitio oficial: `https://cilium.io/`.

[9]**Falco** es un motor de detección de intrusiones de código abierto que utiliza eBPF para monitorear comportamientos anómalos en tiempo real a nivel de sistema. Sitio oficial: `https://falco.org/`.

[10]**Hubble** es una herramienta de observabilidad para redes nativas en Kubernetes, construida sobre Cilium y eBPF, que permite inspeccionar flujos, políticas y métricas de red. Sitio oficial: `https://cilium.io/hubble/`.

[11]**Katran** es un balanceador de carga L4 de alto rendimiento desarrollado por Meta, que utiliza eBPF para procesar tráfico a gran escala con eficiencia. Repositorio oficial: `https://`

para construir soluciones de monitoreo, filtrado y balanceo de carga de nueva generación.

En conjunto, eBPF y XDP redefinen la relación entre el kernel y la red, trasladando parte de la inteligencia de control al propio plano de datos y habilitando la integración con plataformas de telemetría programable y SmartNICs, como se explora en la siguiente sección.

Actualmente, eBPF ha trascendido el ámbito académico para convertirse en una tecnología de referencia en la industria del *cloud computing*. Empresas como Netflix (TechBlog, 2021), Meta (Chauhan, 2025) y Google (Google Cloud Blog, 2024) la utilizan para trazado de rendimiento, análisis de latencia y gestión dinámica de políticas de seguridad.

Cloudflare (Blog, 2022) emplea eBPF en su infraestructura global para filtrar tráfico malicioso en el perímetro de red, mientras que AWS (Amazon Web Services) (Services, 2023) lo integra en su servicio *Elastic Load Balancing* y en su capa de observabilidad VPC Flow Logs. Asimismo, plataformas de código abierto como Cilium se basan en eBPF para proveer observabilidad, control de tráfico y detección de intrusiones en entornos de contenedores (Sharma y Nadig, 2024; Gregg, 2019).

Cilium utiliza eBPF para proveer control de tráfico, seguridad y observabilidad dentro de clústeres Kubernetes, reemplazando el uso de *iptables* por mecanismos de filtrado y balanceo de carga más eficientes. Por su parte, Falco implementa un sistema de detección de intrusiones en tiempo real (IDS) basado en eBPF, capaz de identificar comportamientos anómalos en contenedores y procesos del sistema con bajo impacto en el rendimiento. (Cilium Project, 2025)

La adopción de eBPF también ha alcanzado el ámbito del hardware programable. Algunos fabricantes de interfaces de red, como Netronome (Netronome Systems, Inc., 2018), Intel (I. Corporation, 2022) y NVIDIA (N. Corporation, 2020), incorporan soporte para programas eBPF *en modo offload*, ejecutados directamente en la SmartNIC. Esta capacidad permite trasladar parte del análisis de tráfico y la recolección de métricas al hardware de red, reduciendo la carga del procesador principal y acercando la inteligencia al plano de datos, lo que resulta crítico para redes de alta capacidad y baja latencia (Miano y cols., 2024).

2.3. Integración con aprendizaje automático supervisado

Esta sección se basa fuertemente en el trabajo de Boutaba et al. (Boutaba y cols., 2018), quienes presentan una revisión exhaustiva sobre la aplicación de aprendizaje automático en la gestión y telemetría de redes. A partir de dicho marco, complementado con otras referencias que se explicitan en el texto, se sintetizan aquí los conceptos y enfoques más relevantes relacionados con el aprendizaje supervisado y su integración con arquitecturas programables.

`github.com/facebookincubator/katran`.

El aprendizaje automático supervisado se ha consolidado como una de las herramientas más efectivas para potenciar los sistemas de telemetría y gestión de redes de alta capacidad(Miano y cols., 2024; Jiang y cols., 2021). Este enfoque entrena modelos predictivos a partir de conjuntos de datos etiquetados, de modo que el sistema aprende las relaciones entre variables observadas y estados de la red, permitiendo anticipar comportamientos o anomalías antes de que impacten el servicio. La disponibilidad creciente de datos de telemetría y el poder computacional de los dispositivos modernos han hecho del aprendizaje supervisado un componente clave de la gestión inteligente de redes (Boutaba y cols., 2018). En este contexto, la telemetría programable basada en eBPF, P4 o INT proporciona un flujo continuo de observaciones que puede alimentar modelos predictivos ejecutables en tiempo (casi) real.

Los datos recolectados mediante mecanismos programable constituyen una base ideal para aplicar aprendizaje supervisado. Entre las aplicaciones más relevantes se destacan las siguientes:

- **Detección de anomalías y seguridad:** los modelos basados en árboles, como *RF* y *XGBoost*, permiten identificar patrones de tráfico anómalos asociados a ataques de denegación de servicio, escaneos de puertos o comportamientos inusuales.
- **Predicción de congestión y latencia:** al modelar la relación entre métricas de tráfico (pérdidas, ACKs duplicados, tamaño de ventana, throughput) y estados de congestión, los modelos supervisados facilitan la gestión proactiva de recursos en redes de datacenter y entornos virtualizados.
- **Optimización del rendimiento:** técnicas de regresión y árboles de decisión permiten estimar métricas de rendimiento, como throughput o pérdida esperada en función de las condiciones observadas, habilitando ajustes dinámicos en el plano de control y la planificación de recursos de transporte.

Los modelos supervisados constituyen una buena base de aplicaciones de aprendizaje automático en gestión de redes, dado que permiten aprender relaciones no lineales entre métricas observadas y estados de la red a partir de datos históricos (Boutaba y cols., 2018). Entre los algoritmos más empleados se destacan los siguientes:

- **Árboles de decisión y RF:** ampliamente utilizados para la clasificación de tráfico y detección de anomalías, gracias a su capacidad de manejar variables heterogéneas y capturar interacciones complejas entre atributos sin requerir supuestos estadísticos previos.
- **XGBoost:** una implementación optimizada de *Gradient Boosted Decision Trees* (Chen y Guestrin, 2016), que combina alta precisión con eficiencia computacional. Estudios reportados muestran que este tipo de modelos

alcanzan precisiones superiores al 90 % en tareas de clasificación de tráfico (Boutaba y cols., 2018), lo que motiva su adopción en el presente trabajo como modelo base para clasificación y regresión.

- **Modelos de regresión lineal y regularizada:** empleados para estimar métricas continuas de desempeño. Si bien ofrecen interpretabilidad, suelen ser menos efectivos frente a relaciones no lineales complejas.

Cabe señalar que, si bien técnicas más avanzadas como las redes neuronales recurrentes (RNN) o LSTM han sido exploradas en entornos de predicción de tráfico, su costo de entrenamiento y su necesidad de grandes volúmenes de datos etiquetados las hacen menos adecuadas para escenarios de telemetría en tiempo real sobre SmartNICs o planos de datos programables. (Jiang y cols., 2021)

El uso de aprendizaje supervisado en telemetría ofrece una serie de ventajas significativas. En primer lugar, permite transformar datos brutos de tráfico en información útil para la toma de decisiones, anticipando problemas antes de que impacten en los usuarios finales. Además, mejora la precisión en la detección de anomalías en comparación con enfoques puramente estadísticos, lo que habilita una identificación más temprana de comportamientos inusuales en la red(Wang y cols., 2021). Otra ventaja importante es la capacidad de optimizar dinámicamente parámetros de red en función de predicciones confiables, contribuyendo a una gestión más eficiente y proactiva de los recursos disponibles.

Sin embargo, también existen limitaciones relevantes. Una de ellas es la necesidad de contar con datos etiquetados, ya que la calidad de los modelos supervisados depende directamente de la disponibilidad de conjuntos de entrenamiento representativos y correctamente anotados (Zhang y cols., 2019). A esto se suma la dificultad de adaptación a entornos dinámicos: los modelos deben actualizarse de manera frecuente para ajustarse a cambios en el tráfico o en la topología de red, lo que implica un esfuerzo adicional en mantenimiento y reentrenamiento (Nguyen y Armitage, 2008). Finalmente, el consumo de recursos constituye otra limitación importante, dado que el entrenamiento de modelos complejos puede demandar una capacidad computacional significativa, planteando retos en escenarios de procesamiento en tiempo real (Mijumbi y cols., 2016).

En síntesis, la integración del aprendizaje automático supervisado con la telemetría programable ofrece un camino sólido hacia sistemas de monitorización inteligentes, capaces de anticipar condiciones adversas y optimizar el rendimiento de redes de alta capacidad. Si bien los desafíos asociados a la generación de datasets[12] y a la adaptabilidad de los modelos siguen presentes, la literatura reciente apunta a que estos enfoques representan un cambio de paradigma respecto a la monitorización reactiva tradicional.

[12] Es una colección de datos organizada, a menudo en formato de tabla, donde cada fila es un registro y cada columna es una variable.

2.4. El rol de las SmartNICs

Las tarjetas de interfaz de red (Network Interface Cards (NICs)) han sido tradicionalmente dispositivos pasivos cuya función principal consistía en transmitir y recibir paquetes entre el medio físico y la pila de protocolos del sistema operativo. Sin embargo, la evolución del tráfico en centros de datos, el despliegue de arquitecturas en la nube y la creciente demanda de aplicaciones sensibles a la latencia evidenciaron las limitaciones de este modelo. Estas limitaciones impulsaron la aparición de un nuevo paradigma: las *SmartNICs*, tarjetas de red inteligentes que integran capacidades de procesamiento, memoria y programabilidad avanzadas, y que constituyen hoy un componente estratégico en redes de alta capacidad (Kfoury y cols., 2024; NVIDIA Blog, 2023).

La transición hacia SmartNICs fue progresiva. Las primeras NICs incorporaron funciones de *offload* específicas, como el cálculo de checksums o la segmentación TCP (*TCP Segmentation Offload*), seguidas por capacidades más avanzadas como el manejo de colas múltiples y la gestión de túneles encapsulados (Virtual Local Area Network (VLAN), Network Virtualization using Generic Routing Encapsulation (NVGRE), Generic Network Virtualization Encapsulation (Geneve)). Estos mecanismos buscaban reducir la carga de procesamiento del CPU anfitrión, permitiendo que los servidores dedicaran más recursos a las aplicaciones.

El salto cualitativo se produjo con la integración de procesadores programables en las tarjetas, habilitando que tareas tradicionalmente ejecutadas en el host se realizaran directamente en el dispositivo. Así nacieron las SmartNICs, equipadas con núcleos Advanced RISC Machines (ARM)[13] o Reduced Instruction Set Computing - Five (RISC-V)[14], memorias locales y pipelines reconfigurables basados en el modelo *match–action*[15](Bosshart y cols., 2014). Estas arquitecturas surgieron en respuesta al crecimiento de los centros de datos a escala hiper, donde la flexibilidad y el procesamiento distribuido se volvieron esenciales (Trenton Systems, 2023).

Las SmartNICs pueden considerarse sistemas en chip especializados. Incorporan procesadores embebidos que ejecutan firmware o programas definidos por el usuario, memorias Dynamic Random Access Memory (DRAM) o Static Random Access Memory (SRAM) para almacenamiento temporal y aceleradores dedicados para tareas de red. Además, disponen de interfaces de alta velocidad (Peripheral Component Interconnect Express (PCIe), Non-Volatile Memory express (NVMe)) que les permiten comunicarse con el servidor host mediante acceso directo a memoria (Direct Memory Access (DMA)).

Un ejemplo representativo es la familia *Netronome Agilio CX*, que integra decenas de microprocesadores NFP diseñados para la clasificación y manipulación de flujos de red (Netronome Systems, Inc., 2019). Estas arquitecturas posibilitan

[13] https://www.arm.com/architecture

[14] https://riscv.org/

[15] El modelo *match–action* define el procesamiento de paquetes como una secuencia de etapas donde cada una compara campos del encabezado con reglas almacenadas en tablas (*match*) y ejecuta las acciones correspondientes (*action*), como reenviar, modificar o descartar el paquete

la ejecución de programas eBPF/XDP u otros pipelines de telemetría directamente en la tarjeta, alcanzando niveles de eficiencia imposibles de lograr en el CPU central.

La figura 2.3 indica como en el espacio de usuario se desarrolla el código en C, que se compila con `clang`[16] a bytecode eBPF y se carga al kernel mediante herramientas como `libbpf` o `bpftool`[17]. En el kernel, el verificador eBPF comprueba la seguridad del programa antes de su ejecución dentro de la máquina virtual embebida.

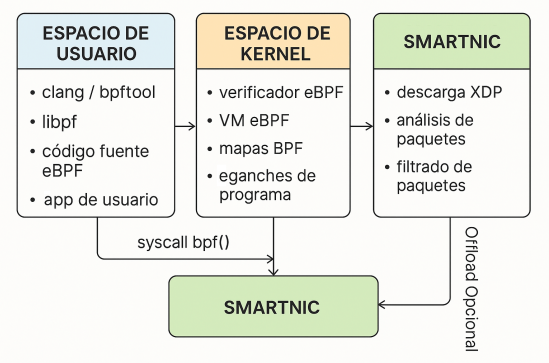

Figura 2.3: Arquitectura colaborativa de eBPF entre el espacio de usuario, el kernel y la SmartNIC.

Las figuras 2.4 y 2.5 nos muestran el flujo de recepción con y sin bytecode eBPF cargado en XDP. En general los paquetes entrantes son depositados en el *RX ring*[18] del controlador de la NIC. Luego si correspondiera el programa XDP asociado se ejecuta antes de que el paquete sea copiado a la estructura `sk_buff`[19], lo que permite tomar decisiones inmediatas de filtrado, redirección o reenvío, en caso de no haber programa cargado el flujo continua según 2.4 con el stack de red. Por otro lado la integración del driver y del hardware genera diferentes modos en que el programa puede ejecutarse, concretamente tres:

- **Generic:** ejecutado en la pila de red tradicional, sin soporte nativo del driver.
- **Native:** ejecutado dentro del controlador de la NIC en el kernel.

[16] Clang es un compilador front end para los lenguajes de programación C, C++, Objective-C, Objective-C++. https://clang.llvm.org/

[17] Es una herramienta de línea de comandos de Linux que se utiliza para inspeccionar y administrar objetos de eBPF. https://www.kernel.org/doc/html/latest/bpf/bpftool.html

[18] Es un búfer de memoria en una tarjeta de red (NIC) que almacena los datos que se están recibiendo.

[19] Es una estructura de datos del subsistema de red del kernel de Linux. Sirve como contenedor fundamental para los paquetes de red a medida que atraviesan las distintas capas de la pila de red, desde la capa física hasta los protocolos de aplicación.

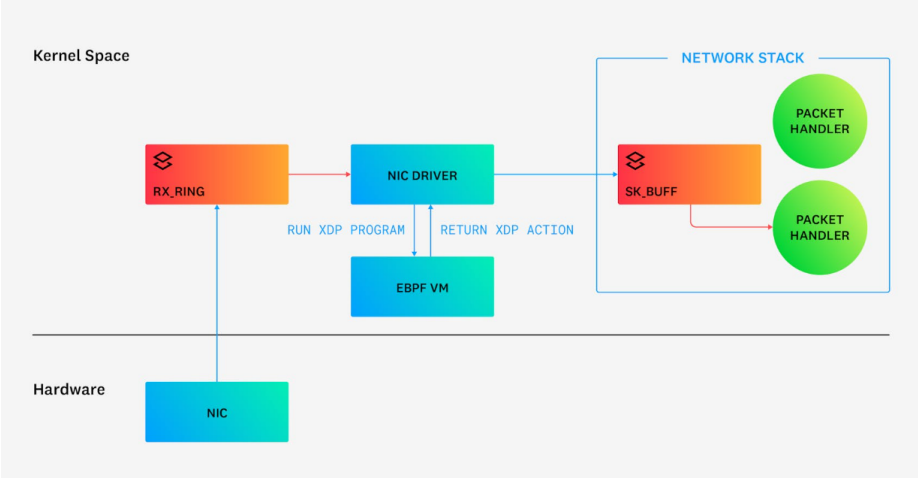

Figura 2.4: Flujo de paquetes en el kernel sin XDP (DataDog, 2023).

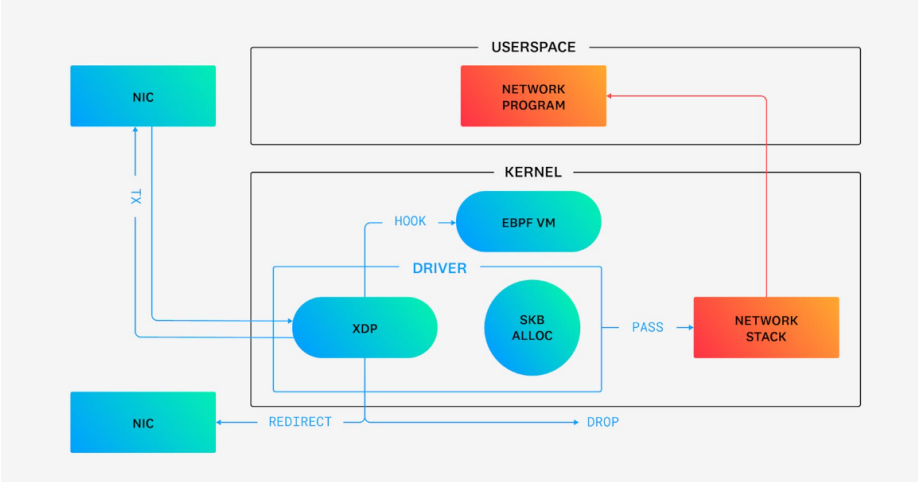

Figura 2.5: Flujo de paquetes en el kernel con XDP (DataDog, 2023).

- **Offloaded:** ejecutado directamente en la SmartNIC, desplazando la ejecución al plano de datos del dispositivo.

En el modo *offload*, la SmartNIC procesa los paquetes de forma autónoma, reduciendo el intercambio de contexto con el CPU y mejorando el rendimiento global del sistema (Høiland-Jørgensen y cols., 2018).

Las SmartNICs amplían significativamente las capacidades de una NIC convencional. Entre sus principales funciones se destacan:

- **Procesamiento de paquetes en línea:** permiten analizar, modificar y reenviar paquetes directamente en la tarjeta antes de entregarlos al sistema operativo.
- **Seguridad y aislamiento:** integran aceleradores para cifrado/descifrado (Transport Layer Security (TLS), Internet Protocol Security (IPsec)) y microsegmentación de flujos.
- **Virtualización y multitenancy:** implementan Single Root I/O Virtualization (SR-IOV) y túneles de red (Geneve, NVGRE), habilitando redes virtuales con aislamiento a nivel de hardware.
- **Telemetría avanzada:** ejecutan programas eBPF o P4 para recolectar métricas en tiempo real desde el plano de datos, minimizando la latencia y el uso de CPU.

Su adopción en la industria refleja su madurez tecnológica. En la nube pública, proveedores como *AWS Nitro*, *Microsoft Azure SmartNIC* y *Google Cloud* emplean estas tarjetas para virtualización, telemetría y seguridad, reduciendo significativamente la carga del procesador (Infinite Networks, 2023). En telecomunicaciones, las SmartNICs se integran en arquitecturas Software Defined Networking (SDN) y Network Function Virtualization (NFV) para desplegar funciones virtualizadas con monitoreo en tiempo real. En entornos financieros y de High-Performance Computing (HPC), su baja latencia permite procesar millones de eventos o transacciones por segundo (Embedded.com, 2023; Kfoury y cols., 2024).

La sinergia entre SmartNICs y telemetría programable constituye uno de los avances más importantes en la monitorización moderna. Gracias a su programabilidad, es posible implementar pipelines definidos en P4 o ejecutar programas eBPF/XDP directamente en la tarjeta, recolectando métricas a nivel de flujo o paquete con mínima latencia. Esta capacidad habilita un preprocesamiento local de los datos y la transmisión de métricas sintetizadas al colector central, optimizando el ciclo de medición, análisis y respuesta (Miano y cols., 2024).

En el contexto de redes de alta capacidad, donde los enlaces de centros de datos ya superan los 100 Gbps, el volumen de tráfico continúa creciendo de manera sostenida. Según el *Cisco Annual Internet Report*, el tráfico global de centros de datos se duplicó aproximadamente cada dos años durante la última década(Cisco Systems, 2020). En este escenario, las SmartNICs ofrecen ventajas

determinantes: reducen la carga del CPU, permiten ejecutar telemetría directamente en el plano de datos y disminuyen la latencia asociada al monitoreo. Estas capacidades las consolidan como componentes fundamentales en arquitecturas modernas de observabilidad y gestión inteligente de redes, especialmente en entornos que combinan *cloud computing*, 5G y analítica en tiempo real.

Se describe en los capítulos siguientes la implementación de esta arquitectura de forma práctica, desplegando programas eBPF tanto en modo *driver* (ejecución en el kernel del host) como en modo *offload* sobre una SmartNIC Netronome Agilio CX. La comparación entre ambos modos permite cuantificar el efecto real de la descarga de procesamiento (*offload*) en términos de rendimiento, uso de CPU y estabilidad del throughput.

2.5. Limitaciones y desafíos

A pesar de los avances en telemetría programable y en el uso de SmartNICs para la monitorización de redes, persisten limitaciones que condicionan su adopción en entornos productivos. En primer lugar, las SmartNICs continúan enfrentando restricciones de memoria y capacidad de procesamiento. Si bien superan a las NICs tradicionales, la memoria disponible para almacenar métricas y tablas de flujos sigue siendo reducida, y su potencia de cómputo resulta insuficiente para ejecutar algoritmos complejos de aprendizaje automático directamente en la tarjeta.

Otro desafío relevante es la heterogeneidad arquitectónica. Los fabricantes ofrecen Software Development Kits (SDKs), modelos de programación y pipelines distintos, lo que dificulta la portabilidad de aplicaciones y obliga a adaptar el flujo de telemetría a cada plataforma específica. La falta de estandarización limita la interoperabilidad y aumenta el esfuerzo de desarrollo e integración (Kfoury y cols., 2024).

La coexistencia con mecanismos de monitoreo tradicionales —como SNMP, NetFlow o sFlow— agrega complejidad al despliegue. Las organizaciones suelen mantener infraestructuras heredadas, por lo que las soluciones basadas en telemetría programable deben integrarse sin interrumpir los sistemas existentes. Asimismo, la introducción de SmartNICs implica ajustes en la gestión del host, la virtualización y los procesos de seguridad, requiriendo personal especializado para su operación.

El costo asociado a las SmartNICs continúa siendo un factor crítico. Estos dispositivos suelen tener un precio significativamente mayor que las NICs convencionales, y su valor agregado se justifica solo cuando los beneficios en rendimiento y telemetría compensan la inversión inicial. Los costos indirectos —como la capacitación, integración y adaptación de procesos— también impactan en la viabilidad del despliegue.

Desde el punto de vista académico, persisten desafíos abiertos. La generación de *datasets* representativos y correctamente etiquetados sigue siendo una barrera para el desarrollo de modelos de aprendizaje supervisado en redes. Además, la naturaleza dinámica del tráfico exige mecanismos de actualización y reentre-

namiento continuo.

La escalabilidad extrema y la seguridad completan el panorama de desafíos: conforme las velocidades de enlace superan los 400 Gbps, las arquitecturas deben evolucionar para mantener una recolección de métricas sin degradar la latencia, garantizando simultáneamente la protección del plano de datos ante vulnerabilidades potenciales.

2.6. Conclusiones

La revisión presentada evidencia que la telemetría programable —particularmente mediante eBPF y XDP— ha transformado la capacidad de observación en sistemas Linux, habilitando recolección de métricas con granularidad fina y mínima sobrecarga. En paralelo, el auge de las SmartNICs introduce la posibilidad de desplazar parte del procesamiento hacia el plano de datos, reduciendo la carga del host y mejorando la latencia de análisis. Estos avances constituyen una base sólida para el desarrollo de sistemas de monitoreo más eficientes y cercanos al tráfico real.

Sin embargo, las tecnologías actuales aún enfrentan limitaciones técnicas, de interoperabilidad y de adopción que restringen su despliegue masivo. La heterogeneidad arquitectónica, los costos asociados, la integración con sistemas de monitoreo tradicionales y la necesidad de personal especializado son factores que condicionan su incorporación en infraestructuras existentes. Del lado del aprendizaje automático, la disponibilidad de datos representativos y la adaptación a entornos dinámicos continúan siendo desafíos abiertos.

En conjunto, el análisis realizado permite identificar tanto el potencial como las limitaciones del estado del arte. Estas observaciones motivan la propuesta desarrollada en este proyecto, que busca combinar telemetría programable, procesamiento distribuido y modelos supervisados para avanzar hacia un sistema de monitoreo predictivo eficiente y aplicable en redes de alta capacidad.

Capítulo 3

Diseño e Implementación de un Sistema de Telemetría Inteligente sobre SmartNICs

El presente capítulo describe el desarrollo e implementación del sistema de telemetría inteligente propuesto en el marco de este proyecto de grado. Se detalla la producción propia, las decisiones de diseño y las estrategias de integración adoptadas para construir una solución funcional capaz de recolectar, procesar y analizar métricas de red en tiempo real.

El sistema se desarrolló sobre un entorno Linux (kernel versión 6.8) con soporte nativo para eBPF y XDP, utilizando una tarjeta SmartNIC Netronome Agilio CX NFP-4000 como plataforma de experimentación. Este hardware permitió validar la ejecución de programas eBPF tanto en modo *driver* como en modo *offload*, evaluando el impacto de trasladar el procesamiento desde el CPU central hacia el plano de datos de la interfaz de red.

La SmartNIC Netronome Agilio CX (NFP-4000) empleada en este trabajo cuenta con una arquitectura altamente paralela, como indica la figura 3.1, está compuesta por 60 Flow Processing Cores (FPCs) con hasta 8 hilos cooperativos por núcleo (Netronome Systems, Inc., 2018). Los programas eBPF se ejecutan en modo offload simultáneamente sobre múltiples de estos núcleos, procesando diferentes paquetes en paralelo. (Netronome Systems, Inc., 2018)

El diseño general del sistema se basó en una *arquitectura jerárquica de tres niveles* —SmartNIC, kernel y espacio de usuario— que habilita un flujo continuo de recolección, agregación y análisis de métricas. Esta estructura permitió desacoplar las funciones de captura, consolidación y persistencia de los datos, reduciendo la sobrecarga sobre el procesador principal y favoreciendo la modularidad del desarrollo.

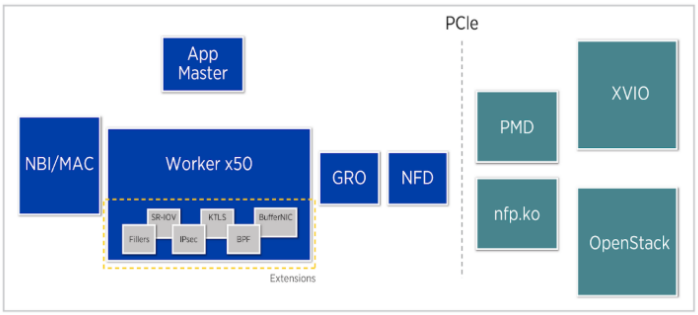

Figura 3.1: Arquitectura de la SmartNIC Agilio CX. (Netronome Systems, Inc., 2018).

La Figura 3.2 ilustra esta arquitectura general. En el plano de recolección, los programas eBPF/XDP ejecutados en la SmartNIC clasifican los paquetes y contabilizan flujos TCP utilizando mapas eBPF de tipo `HASH` y `ARRAY`. A nivel de kernel (plano intermedio), se mantienen los mapas `pinned`[1] en el sistema de archivos `/sys/fs/bpf`, lo que garantiza la persistencia de los datos y su lectura concurrente por procesos en el espacio de usuario. Finalmente, el plano superior implementa los módulos encargados de la recolección, almacenamiento y exposición de métricas.

El sistema implementado, junto con la documentación necesaria para su instalación, configuración y despliegue, se encuentra disponible en el repositorio institucional de la Facultad de Ingeniería, bajo licencia abierta. [2]

3.1. Requerimientos y objetivos específicos

El desarrollo del sistema de telemetría inteligente se estructuró entorno a un conjunto de requerimientos funcionales y no funcionales definidos en base a los objetivos generales del proyecto de grado. Estos requerimientos orientaron las decisiones de diseño y sirvieron como criterios de validación técnica en cada una de las etapas de implementación.

3.1.1. Requerimientos funcionales

Los requerimientos funcionales establecen las capacidades que el sistema debía ofrecer para cumplir su propósito de monitoreo, procesamiento y análisis predictivo de tráfico en tiempo real. Se definieron los siguientes:

[1] En eBPF, el término *pinned map* (mapa anclado) se refiere a una estructura de datos persistente almacenada en el sistema de archivos virtual `/sys/fs/bpf/`. Esto permite que múltiples procesos del espacio de usuario accedan al mismo mapa de forma concurrente y que su contenido se mantenga disponible incluso después de que el programa eBPF haya finalizado.

[2] Repositorio institucional: `https://gitlab.fing.edu.uy/smartlab/monitorizacion-cc`

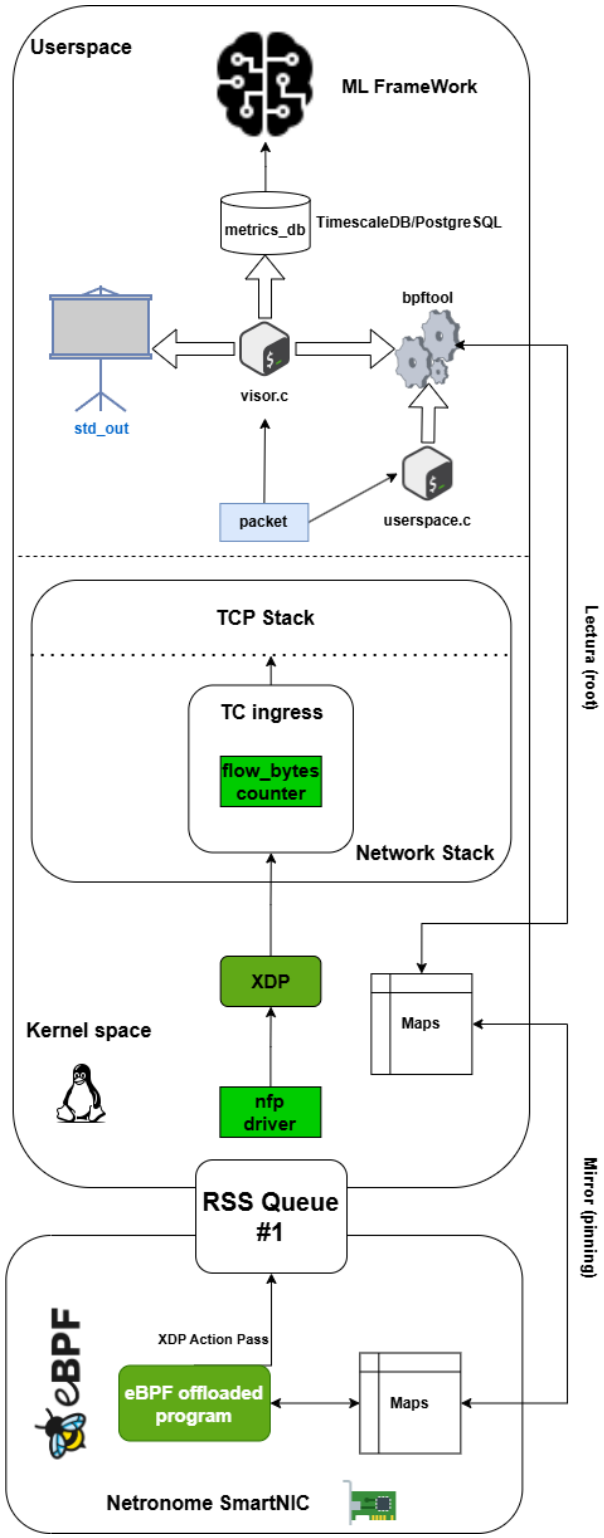

Figura 3.2: Arquitectura jerárquica del sistema de telemetría inteligente.

- **Recolección de métricas de tráfico:** el sistema debe capturar información de red en tiempo real, incluyendo cantidad y tamaño de paquetes, número de flujos activos, flags TCP, tasas de envío y métricas temporales (timestamp y tiempo transcurrido entre paquetes consecutivos).
- **Ejecución en la SmartNIC:** los programas eBPF/XDP deben ejecutarse tanto en modo *driver* como en modo *offload* sobre la tarjeta SmartNIC Netronome Agilio CX NFP-4000.
- **Agregación y persistencia de métricas:** el kernel debe consolidar los datos recolectados, de forma que los módulos de espacio de usuario puedan acceder a ellos de manera concurrente y persistente.
- **Integración con sistemas de almacenamiento:** las métricas recolectadas deben persistirse permitiendo consultas históricas y análisis por intervalos temporales.
- **Exposición de métricas:** el sistema debe exponer las métricas recolectadas de forma que puedan visualizarse.
- **Recolección complementaria del sistema operativo:** el sistema debe registrar métricas del host desde el subsistema `/proc` (memoria, CPU, disco, red).
- **Generación de datasets para aprendizaje automático:** los datos almacenados deben ser exportables en formato Comma-Separated Values (CSV), estructurados y etiquetados según condiciones observadas durante las pruebas experimentales, diferenciando escenarios de congestión y no congestión.

3.1.2. Requerimientos no funcionales

Los requerimientos no funcionales definen las propiedades de calidad del sistema, orientadas al desempeño, eficiencia y escalabilidad. Se establecieron los siguientes:

- **Escalabilidad:** el sistema debe ser capaz de manejar al menos cientos de flujos concurrentes sin pérdida significativa de rendimiento, garantizando la consistencia de las métricas recolectadas.
- **Modularidad y extensibilidad:** la arquitectura debe mantener una separación clara entre los niveles de SmartNIC, kernel y espacio de usuario, permitiendo modificar o reemplazar componentes de forma independiente.
- **Consistencia temporal y persistencia:** las métricas deben registrarse con timestamps unificados, manteniendo la coherencia entre las fuentes de red y del sistema operativo, tanto en los archivos CSV como en la base de datos.

- **Integración continua y automatización:** el sistema debe poder desplegarse mediante uno o varios scripts de automatización que coordinen la carga de programas, la ejecución de recolectores y el arranque de servicios.

3.1.3. Proceso de desarrollo e ingeniería de software

El desarrollo del sistema se llevó a cabo siguiendo un enfoque *incremental–iterativo*, siguiendo las etapas detalladas en la Figura 3.3, organizado en ciclos cortos que combinaron análisis, diseño, implementación y validación experimental. En un principio, dada la naturaleza novedosa del hardware disponible, nos dedicamos a la investigación del mismo para entrar en contacto y tener primeras impresiones. A la vez se hizo una primera iteración donde se implementó un prototipo puramente software del plano de datos, ejecutando eBPF/XDP en el host. En iteraciones posteriores se incorporó el modo *offload* sobre la SmartNIC, la integración con TimescaleDB/Grafana y, finalmente, el pipeline de generación de dataset y entrenamiento de modelos de aprendizaje automático.

Aunque no se implementó una integración continua completa, se automatizaron tareas clave mediante scripts que compilaban los programas eBPF, inicializaban la base de datos y ejecutaban pruebas de carga reproducibles. Este enfoque permitió validar la corrección del sistema tras cambios en el código y facilitó la repetición de experimentos.

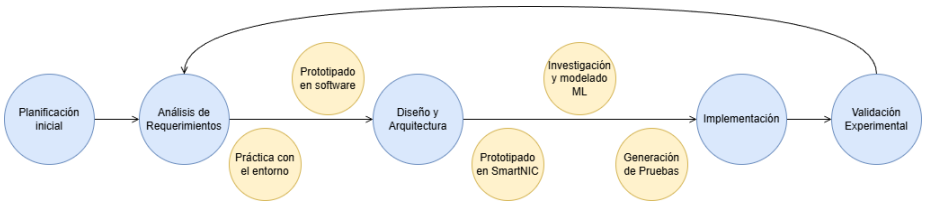

Figura 3.3: Resumen de las principales etapas del proceso de desarrollo (etapas en color amarillo se cumplen una vez).

Tiempos y esfuerzo del proyecto

El desarrollo se extendió a lo largo de aproximadamente 8 meses. La tabla 3.1 resume de manera aproximada la distribución del esfuerzo por fase.

3.2. Diseño de la arquitectura

El sistema propuesto se construyó bajo una arquitectura jerárquica de tres niveles —SmartNIC, kernel y espacio de usuario— que permite distribuir las funciones de telemetría, reducir la carga sobre el CPU y lograr una observabilidad de red continua y de baja latencia. Este diseño sigue el principio de desplazar el procesamiento hacia el plano de datos, manteniendo al mismo tiempo la flexibilidad del espacio de usuario para análisis y persistencia.

Tabla 3.1: Fases y tiempos de esfuerzo aproximados.

Fase	**Tiempo aproximado**
Análisis de antecedentes y definición de requerimientos	3 semanas
Diseño de arquitectura y prototipos iniciales	4 semanas
Implementación y pruebas de la telemetría en SmartNIC	17 semanas
Generación del dataset y entrenamiento de modelos ML	4 semanas
Experimentación, análisis de resultados y redacción	2 semanas

La arquitectura de la Figura 3.2 muestra la estructura general del sistema, que integra la captura de paquetes en la SmartNIC Netronome Agilio CX NFP-4000, la consolidación de métricas en el kernel mediante mapas eBPF y la recolección, persistencia y exposición de datos en el espacio de usuario.

3.2.1. Nivel de recolección de datos (SmartNIC – Plano de datos)

El primer nivel de la arquitectura se ejecuta directamente en la SmartNIC y constituye el plano de datos del sistema. En este nivel se desplegaron los programas eBPF/XDP que interceptan los paquetes en la interfaz física antes de que sean procesados por la pila de red del kernel.

El programa principal, `offload.c`, clasifica los paquetes según su tamaño y protocolo (TCP, User Datagram Protocol (UDP), Address Resolution Protocol (ARP), Ethereum Wire Protocol (ETH), IP), contabilizando el número de paquetes por segundo, numero de paquetes por flag, numero de paquetes por flujo y numero de flags por flujo. Se implementaron tres mapas de tipo ARRAY y 2 de tipo HASH:

- `map_size_count`: registra la cantidad de paquetes en general y los clasifica por tamaño.
  - `0 (PKT_0)`: longitud total menor a 100 bytes.
  - `1 (PKT_1)`: longitud total entre 100 y 200 bytes.
  - `2 (PKT_2)`: longitud total entre 200 y 300 bytes.
  - `3 (PKT_3)`: longitud total entre 300 y 400 bytes.
  - `4 (PKT_4)`: longitud total mayor a 400 bytes.
- `map_arr_rate`: contador global de paquetes usado para calcular la tasa de llegada en pps (paquetes por segundo).

- `tcp_flags_count`: contador global de flags TCP (sin discriminar por flujo), paquetes por tipo de protocolo y ECN-CE en el encabezado IP. Siguiendo el orden en el mapa serían: FIN, SYN, RST, PSH, ACK, URG, ETH_IP, ETH_ARP, TCP, UDP, ECN_CE.
- `flow_bytes`: acumula la cantidad de paquetes transmitidos por flujo.
- `flow_flags_map`: acumula la cantidad de flags y ECN_CE asociadas a cada flujo.

En el plano de datos cada flujo se identifica mediante una estructura clave compuesta por la 5-tupla `(src_ip, dst_ip, src_port, dst_port, protocol)`, lo que permite mantener independencia entre flujos simultáneos y garantizar coherencia en las estadísticas. El siguiente nivel da origen a los *flowlets*.

El programa opera tanto en modo *driver* como en modo *offload*, valiéndose de la capacidad de la SmartNIC Netronome para ejecutar bytecode eBPF directamente en sus procesadores integrados (NFP). Esta característica evita que el CPU central deba procesar cada paquete, descargando de manera efectiva la clasificación y el conteo hacia el hardware especializado.

Debido a las limitaciones del conjunto de instrucciones soportado por el firmware NFP-4000, algunas funciones *helper* de eBPF, como `bpf_map_update_elem()`, no se encuentran disponibles para ejecución en hardware. Por tanto, las operaciones sobre mapas se restringen a búsquedas e incrementos, tal como se detalla en (Netronome Systems, Inc., 2018).

3.2.2. Nivel de agregación local (Kernel)

El kernel actúa como capa intermedia de agregación y sincronización. Aquí se mantienen los mapas eBPF *pinneados* en el sistema de archivos `/sys/fs/bpf`, garantizando su persistencia y la posibilidad de ser accedidos por procesos en el espacio de usuario mediante `bpf_obj_get()`.

La función principal de este nivel es mejorar las métricas recibidas desde la SmartNIC y proveer una interfaz segura de acceso a los datos. Para ello, se construyen otros mapas que viven en el host y favorecen la interacción con los módulos del espacio de usuario, estos son:

- `flow_timer`: Almacena la marca de tiempo del ultimo paquete visto para cada flujo.
- `flow_let`: Almacena el indice del ultimo flowlet identificado para cada flujo.
- `flow_pid`: Almacena el pid del proceso que atiende cada flujo.
- `flow_stats`: Almacena métricas del kernel y la red referidas al propio flujo.

Métricas directas del kernel:

- `snd_cwnd`: Tamaño actual de la ventana de congestión (en segmentos MSS). Indica cuántos segmentos TCP puede enviar sin esperar ACKs.
- `rtt_us`: Round-Trip Time.
- `bytes_acked`: Total acumulado de bytes reconocidos.
- `bytes_retrans`: Total acumulado de bytes retransmitidos.
- `retrans_pkts`: Total acumulado de paquetes retransmitidos.
- `rcv_space`: Ventana de recepción local.
- `snd_wnd`: Ventana de envío actual.
- `snd_wscale/rcv_wscale`: Factores de escala aplicados al tamaño de las ventanas.

Métricas Calculadas:

- `throughput_mbps_x100`: Calculado como

$$\frac{\triangle bytes\_ack}{\triangle t * 10^6}$$

  es la velocidad instantánea del flujo en megabits por segundo x 100.
- `last_update_ns`: Última marca de tiempo en que se actualizó esta entrada.

Métricas Extraídas de los paquetes:

- `next_seq_a/next_seq_b`: Próximo número de secuencia esperado en cada dirección (A→B, B→A).
- `loss_dup_pkts_a/b`: Paquetes duplicados detectados (llegaron con `SEQ < next_seq`).
- `loss_dup_bytes_a/b`: Bytes de payload duplicados.
- `loss_ofo_pkts_a/b`: Paquetes out-of-order (`SEQ > next_seq` se adelantaron).
- `loss_ofo_bytes_a/b`: Bytes fuera de orden.
- `gap_lo_a/b, gap_hi_a/b`: Rango [lo, hi) del último hueco secuencial observado. Permite saber el rango de SEQ pendiente de llenar.

Fue definido otro mapa llamado `flow_tc_bytes`, que contribuye al conteo de bytes por flujos. Éste tiene la función de almacenar la quintupla asociada al flujo y contabilizar los bytes transmitidos en cada uno. Como su nombre indica a este mapa se accede desde un hook en TC ingress y es el único programa cargado en este hook dentro de la arquitectura. Si bien la función que este programa desempeña podría lograrse en el espacio de usuario resulta más eficiente implementarlo antes en la pila TCP.

Se definieron políticas de inactividad de flujos con dos umbrales temporales. En primer lugar, si el intervalo entre paquetes de una misma 5-tupla supera los 5 segundos, el sistema registra un nuevo *flowlet* (en el mapa flow_let) asociado al mismo flujo sin eliminar la entrada existente en los mapas. En segundo lugar, si la inactividad alcanza los 30 segundos, un proceso en espacio de usuario (userspace.c) elimina la entrada correspondiente en los mapas eBPF (`flow_bytes`, `flow_timer`, `flow_pid`, `flow_let`, `flow_flag`, `flow_tc_bytes`, `flow_stats`).

De esta forma, cualquier tráfico posterior con la misma 5-tupla se considera un flujo nuevo. Esta estrategia permite evitar el crecimiento indefinido de las estructuras de datos, preservar la coherencia temporal de las métricas y mantener controlado el uso de memoria en ejecuciones prolongadas.

La elección de tipos de mapas se realizó considerando tanto la naturaleza del dato como las limitaciones de memoria de la SmartNIC:

- `BPF_MAP_TYPE_HASH`: para almacenamiento dinámico de flujos con claves complejas (5-tupla).
- `BPF_MAP_TYPE_ARRAY`: para métricas globales, como conteo total de paquetes o distribución por tamaño.

3.2.3. Nivel de procesamiento y persistencia (Espacio de usuario)

El nivel superior de la arquitectura corresponde al espacio de usuario, donde las métricas recolectadas son leídas, procesadas, persistidas y expuestas a sistemas externos. Este nivel se compone de tres módulos principales:

- **Visor de métricas (`visor.c`):** encargado de leer los mapas eBPF pinneados, mostrar las métricas en tiempo real y registrar los datos en archivos CSV y en la base de datos TimescaleDB utilizando la librería `libpq`.
- **Recolector del sistema (`get_sys_metrics.c`):** extrae métricas de los archivos del subsistema `/proc`, como memoria, CPU, disco, red, TCP y UDP. Registra los datos en CSV y simultáneamente los inserta en la base de datos.
- **Agente espacio de usuario (userspace.c):** captura los paquetes TCP, inicializa los flujos al ver un paquete de tipo SYN, normaliza la clave del flujo, actualiza métricas y contadores calculables, asocia flujos con los procesos que los llevan a cabo, asigna identificadores a los subflujos, y se hace cargo del borrado de los flujos inactivos.
- **Grafana:** expone las métricas recolectadas a través de un endpoint HTTP local.

El flujo de trabajo en este nivel sigue un esquema periódico de consulta que por defecto es 0.1 segundos.

Las inserciones en TimescaleDB se realizan con la función `PQexecParams()`, construyendo dinámicamente las sentencias SQL a partir de los valores recolectados. Esta aproximación permite garantizar consistencia entre los datos almacenados en CSV y los registrados en la base de datos, además de ofrecer mayor control sobre la estructura de las tablas y las consultas analíticas.

En esta capa también se ejecutan los componentes analíticos que procesan las métricas recolectadas, incluyendo los módulos de aprendizaje automático que permiten clasificar o predecir estados de la red a partir de las series temporales obtenidas.

En la tabla A.1 podemos encontrar un resumen general de las métricas que consideramos principales.

3.2.4. Flujo de información y sincronización

El flujo de información del sistema se organiza siguiendo un modelo jerárquico de tres niveles, en el que cada componente cumple una función específica dentro del ciclo de adquisición, agregación y análisis de métricas.

En el primer nivel, la SmartNIC actúa como punto inicial de captura. Los programas eBPF/XDP interceptan los paquetes directamente en la interfaz de red y actualizan los mapas de métricas con información agregada por flujo.

En el segundo nivel, el kernel mantiene los mapas eBPF accesibles y actualizados, sirviendo como capa de agregación intermedia. Los mapas se encuentran *pinneados* en el sistema de archivos `/sys/fs/bpf`, lo que permite que múltiples procesos en el espacio de usuario puedan leerlos de forma concurrente sin necesidad de copias adicionales. Además, el kernel sincroniza los datos y preserva la consistencia estructural entre las métricas generadas en la SmartNIC y las consultas posteriores.

Finalmente, en el tercer nivel, los módulos en espacio de usuario ejecutan las tareas de lectura, procesamiento y persistencia de métricas.

La sincronización temporal entre los distintos niveles se implementó mediante el uso de relojes monotónicos [3] (`clock_gettime(CLOCK_MONOTONIC)`), lo que asegura la coherencia entre las fuentes de medición y permite correlacionar eventos de red y del sistema dentro de una misma escala temporal.

En conjunto, este flujo de información jerárquico garantiza una coordinación precisa entre la SmartNIC, el kernel y el espacio de usuario.

[3] A diferencia del reloj de tiempo real (CLOCK_REALTIME), el reloj monotónico garantiza estricta monotonía: su valor solo puede incrementarse y no está sujeto a modificaciones externas tales como sincronización NTP, cambios manuales de hora o ajustes de zona horaria. Esta propiedad lo vuelve adecuado para la medición de intervalos entre paquetes, el cálculo del *inter-arrival time*, la detección de *flowlets* y la expiración de flujos inactivos. El uso de clock_gettime(CLOCK_MONOTONIC) asegura que todos los deltas temporales sean consistentes, evitando valores negativos o discontinuidades que afectarían la precisión de las métricas de telemetría recolectadas.

Entorno de ejecución y dependencias

El entorno de ejecución se basó en una distribución Linux compilado con soporte nativo para eBPF y XDP. El hardware utilizado correspondió a una tarjeta SmartNIC Netronome Agilio CX NFP-4000, disponible en el proyecto de investigación SmartLAB de la Facultad de Ingeniería, empleada para validar la ejecución de programas en modo *offload*.

El entorno de software incluyó las siguientes dependencias:

- **Compilación eBPF:** `clang`, `llvm`, `libbpf-dev` y `bpftool`.
- **Persistencia:** `PostgreSQL 14` con la extensión TimescaleDB.
- **Visualización:** Grafana.
- **Automatización y monitoreo:** `bash`, `python3`, `iperf3`, `libpq-dev`.

Todos los componentes fueron centralizados en el repositorio insitucional[4], el cual incluye los programas eBPF, los módulos en C para el espacio de usuario y los scripts de ejecución.

3.2.5. El uso de TimescaleDB y Grafana

Las métricas de telemetría generadas por sistemas de red y por el sistema operativo forman *series temporales*: secuencias de datos indexadas por tiempo, producidas a intervalos regulares y consultadas principalmente mediante ventanas temporales. Este patrón está ampliamente documentado en arquitecturas de monitorización modernas, donde la telemetría se describe como un flujo continuo de mediciones cuyo eje principal de indexación es el tiempo (Systems, 2019; et al., 2022; Barroso, Clidaras, y Hölzle, 2013). Dado que estas métricas presentan altas tasas de inserción y requieren consultas frecuentes por rango temporal, la elección del motor de persistencia resulta un componente crítico en la arquitectura.

Para satisfacer este requerimiento de persistencia orientada a series temporales se consideraron distintas alternativas:

- **PostgreSQL "puro"**: utilizar exclusivamente un motor relacional sin extensiones específicas para series temporales, gestionando manualmente el particionamiento por tiempo, los índices asociados y las políticas de retención. Este enfoque incrementa el costo operativo y requiere una configuración compleja para mantener un rendimiento adecuado (*PostgreSQL Documentation: Table Partitioning*, 2023).
- **Bases de datos de series temporales dedicadas**: por ejemplo InfluxDB (*InfluxDB Documentation*, 2023) o el motor Time Series Database (TSDB) de Prometheus (*The Prometheus Time Series Database*, 2023), diseñadas para métricas de monitorización, pero que introducen un nuevo

[4] https://gitlab.fing.edu.uy/smartlab/monitorizacion-cc

stack tecnológico, un modelo de datos no relacional y lenguajes de consulta distintos a SQL.

- **Extensiones de PostgreSQL para series temporales**, en particular *TimescaleDB* (*TimescaleDB Whitepaper*, 2020), que incorpora soporte nativo para series temporales manteniendo compatibilidad total con SQL, con el ecosistema de PostgreSQL y con las bibliotecas existentes (*TimescaleDB Documentation: Hypertables*, 2023).

El uso de PostgreSQL sin extensiones se descartó porque delega en el desarrollador tareas críticas como el particionamiento temporal, la agregación continua de métricas y el manejo del ciclo de vida de los datos, lo que incrementa la complejidad y reduce la eficiencia. Por otra parte, las TSDBs dedicadas implican incorporar un motor adicional en la arquitectura y adoptar un modelo de consulta distinto, dificultando la integración con otros componentes que ya utilizan PostgreSQL.

TimescaleDB, en cambio, se presenta como una extensión de PostgreSQL orientada específicamente a datos de series temporales (*TimescaleDB Whitepaper*, 2020). Su principal abstracción, la *hypertable*, divide los datos automáticamente en *chunks* basados en el tiempo y en dimensiones adicionales, optimizando tanto la inserción como las consultas por rango temporal (*TimescaleDB Documentation: Hypertables*, 2023). Además, provee funcionalidades avanzadas como compresión nativa, agregados continuos y políticas automáticas de retención, lo cual resulta especialmente adecuado para sistemas de monitorización con alto volumen de datos temporales (*TimescaleDB Native Compression*, 2023).

En el contexto de este proyecto, TimescaleDB aporta las siguientes ventajas concretas:

- **Consultas expresivas en SQL**: permite consultar métricas mediante SQL estándar, combinando filtros temporales con filtros por flujo, interfaz o host, lo que evita adoptar nuevos lenguajes de consulta.
- **Escalabilidad y eficiencia**: el particionamiento automático en *chunks* y las estructuras optimizadas para series temporales mejoran significativamente el rendimiento de consultas por ventanas temporales y reducen el costo de almacenamiento gracias a la compresión.
- **Integración nativa con Grafana**: TimescaleDB es soportado directamente por Grafana como fuente de datos, facilitando la creación de paneles interactivos y la exploración visual de series temporales (*TimescaleDB as a Grafana Data Source*, 2023; *Grafana Documentation*, 2023).

La elección de **Grafana** se justifica porque es una de las plataformas estándar en la industria para la visualización de métricas y series temporales. Su uso está alineado con las recomendaciones de la literatura en ingeniería de confiabilidad de sitios (SRE), donde se enfatiza la importancia de disponer de paneles dinámicos para analizar tendencias, correlacionar eventos y detectar anomalías

en tiempo real (Beyer, Jones, Petoff, y Murphy, 2016). Asimismo, Grafana permite integrar múltiples fuentes de datos, ejecutar consultas en tiempo real y construir paneles interactivos de forma flexible, características necesarias para el análisis de las métricas recolectadas en este proyecto.

Por estos motivos, y dado que satisface adecuadamente el requerimiento de persistir, consultar y visualizar series temporales de métricas, en este proyecto se seleccionaron **TimescaleDB** como base de datos de series temporales y **Grafana** como plataforma de visualización interactiva.

3.2.6. Integración con TimescaleDB y Grafana

Cada métrica de red o del sistema operativo se asocia a un timestamp y a un conjunto de claves de identificación. Esto aprovecha el modelo de *hypertables* para series temporales de *TimescaleDB*.

Las inserciones se realizan mediante `PQexecParams()`, construyendo dinámicamente las sentencias SQL a partir de los valores recolectados. Esta técnica ofrece control sobre el esquema de almacenamiento y asegura integridad entre los registros CSV y las tablas de la base.

Los dashboards en Grafana combinan métricas del plano de datos (tráfico, flujos, flags TCP) con métricas del sistema operativo (CPU, memoria, disco, red), proporcionando una vista integral del rendimiento del entorno de ejecución.

3.2.7. Scripts de ejecución y automatización

La automatización del sistema se realizó mediante una serie de scripts en Bash y Python, que coordinan la compilación, carga y ejecución de los programas y servicios. El script principal, `run.sh`[5], ejecuta las siguientes tareas:

1. Compila los programas eBPF con `clang`.
2. Carga los objetos en el kernel o en la SmartNIC según el modo (driver u offload).
3. Inicializa los mapas pinneados y los recolectores de métricas.
4. Lanza los procesos de monitoreo (`visor.c`, `get_sys_metrics.c`).

Estos scripts junto a la ejecución de `userspace.c` permiten desplegar todo el sistema, reduciendo los tiempos de configuración y asegurando la repetibilidad de los experimentos.

3.3. Recolección de datos y etiquetado para aprendizaje automático

Una vez implementada la infraestructura de telemetría, se desarrolló una etapa experimental orientada a la recolección y preparación de datos para el

[5]Este programa tiene su análogo llamado `run_drv.sh` que carga el sistema en modo driver.

entrenamiento de modelos de aprendizaje automático supervisado. Esta fase tuvo como objetivo generar un conjunto de datos representativos de diferentes condiciones de tráfico, incluyendo estados de operación normal y de congestión, a partir de las métricas recolectadas por los programas eBPF y los módulos de usuario.

Para la generación del tráfico se utilizó la herramienta iperf3, configurada para simular flujos TCP sostenidos bajo distintos niveles de carga. Se definieron escenarios controlados con 32 flujos paralelos, la variación del tamaño de ventana de emisión y las tasas de transmisión, con el fin de inducir condiciones de congestión en determinados intervalos.

El entorno de pruebas se desplegó sobre una topología mínima compuesta por dos nodos conectados directamente: un emisor y un receptor. La SmartNIC Netronome Agilio CX se utilizó en el nodo receptor, ejecutando el programa eBPF en modo *offload* para capturar y clasificar el tráfico entrante. Las mediciones de tráfico se complementaron con la recolección periódica de métricas del sistema operativo, lo que permitió correlacionar el estado del host con las métricas de red.

La SmartNIC cuenta con dos puertos físicos, por lo que lógicamente fueron separadas en dos espacios de nombre diferentes, de manera que el stack de procesamiento también fuera diferente y pudiese operar como receptor y como emisor el mismo host. Sobre uno de los puertos es que se apoya toda la arquitectura y desde el otro puerto se lanza el tráfico. Ambos puertos se encuentran unidos físicamente por un cable de Fibra Óptica (Fiber Optic) (FO).

El conjunto de métricas recolectadas incluye tanto variables del plano de datos como del sistema operativo. Entre las principales se encuentran:

- **Métricas de red (nivel eBPF):** tamaño de paquete, cantidad de paquetes por flujo, bytes transmitidos, flags TCP (SYN, ACK, FIN, RST), tasa de llegada de paquetes (*packet arrival rate*) y tiempo transcurrido entre paquetes (*delta*).

- **Métricas de kernel y sistema operativo:** utilización de CPU, memoria total y libre, tráfico agregado por interfaz, número de sockets activos, contadores de retransmisiones TCP, estadísticas de disco y carga promedio del sistema.

Las métricas provenientes de los programas eBPF fueron leídas desde los mapas, mientras que las del sistema operativo se obtuvieron desde los archivos del subsistema `/proc`.

3.3.1. Estructura del dataset

Cada registro del *dataset* final corresponde a una observación temporal de un flujo o conjunto de flujos dentro de una ventana de muestreo de 0.1 segundos, generada mediante la captura continua en el plano de datos del sistema

eBPF/XDP. Las columnas incluidas representan tanto los identificadores de flujo como métricas derivadas de tráfico y de protocolo, organizadas del siguiente modo:

- **Identificadores del flujo:** `src_ip`, `dst_ip`, `src_port`, `dst_port`, `protocol`. Permiten distinguir unívocamente cada flujo TCP o UDP según la 5-tupla canónica.
- **Control temporal y de muestreo:** `ts` (timestamp de captura), `last_timestamp_ns` (marca temporal del último paquete) y `delta_ns` (diferencia temporal entre paquetes consecutivos).
- **Métricas de tráfico:** `packets` y `bytes`, acumulativas por flujo dentro de cada ventana de observación. Se complementan con `throughput`, que expresa la tasa efectiva de transferencia de datos.
- **Indicadores del flujo y segmentación:** `flow_let` (identificador de subflujo o *flowlet*), y `pid` (identificador del proceso asociado, en caso de captura local).
- **Flags del protocolo TCP:** `fin`, `syn`, `rst`, `psh`, `ack`, `urg`. Estas permiten caracterizar el estado del flujo y detectar eventos como inicios, finalizaciones o retransmisiones anómalas.
- **Métricas derivadas:** `loss_est_pkts` (pérdida de paquetes estimada), `dup_acks_est` (ACKs duplicados detectados), empleadas como indicadores de congestión.
- **Etiqueta de control de congestión:** `ce`, que marca la presencia del bit ECN-CE en los paquetes, utilizado como variable objetivo o etiqueta supervisada en los modelos de aprendizaje.

El dataset se diseñó para ser procesado posteriormente mediante notebooks de Python, permitiendo análisis exploratorios y entrenamiento de modelos utilizando librerías como `pandas`, `scikit-learn` y `xgboost`. La estructura de los datos asegura compatibilidad con modelos supervisados de clasificación binaria.

3.3.2. Procesamiento y validación de los datos

Una vez completada la recolección, los datos fueron procesados para eliminar registros duplicados, normalizar unidades y verificar la coherencia temporal. Se descartaron observaciones incompletas o inconsistentes, manteniendo únicamente las muestras con información válida de `cwnd`, `rtt`, `bytes_retrans` y `throughput`.

La validación del dataset se realizó mediante análisis estadístico de las variables y visualización de distribuciones para detectar valores atípicos. Asimismo, se calcularon correlaciones entre las métricas de red y de sistema, confirmando la coherencia de las mediciones recolectadas por la SmartNIC y los módulos de usuario.

El resultado final fue un conjunto de datos estructurado y consistente, adecuado para el entrenamiento y evaluación de modelos de clasificación supervisada orientados a la predicción de congestión en redes de alta capacidad.

3.4. Modelado y entrenamiento del sistema predictivo

La etapa final del proyecto consistió en el diseño, entrenamiento y evaluación de modelos de aprendizaje automático supervisado, orientados a la predicción de condiciones de congestión en redes de alta capacidad. La presente sección describe el diseño, entrenamiento y evaluación de dos modelos de aprendizaje automático con objetivos complementarios entrenados a partir de las métricas recolectadas por el sistema de telemetría implementado

- Clasificador de Congestión Explícita (ECN-CE): detecta incrementos en los paquetes marcados con ECN-CE, señal temprana de congestión en la red.
- Regresor de Intensidad de Flujo: estima la cantidad de paquetes transmitidos en la siguiente ventana temporal.

Ambos modelos emplean técnicas basadas en árboles de decisión (XGBoost) debido a su robustez frente a ruido. Con estos modelos se pudo demostrar capacidad para anticipar degradaciones de rendimiento a partir de información histórica de tráfico y realizar una predicción de parámetros.

3.4.1. Selección de variables y preprocesamiento

Para evaluar la relación entre cada característica y la variable objetivo se utilizó el coeficiente de correlación lineal de Pearson [6], denotado como $\rho(x, y)$, que mide la asociación lineal entre ambas variables. Se filtraron las variables por su correlación con el objetivo, conservando únicamente aquellas con $|\rho| < 0{,}9$ para reducir dependencia directa con y_t y evitar *data leakage*.

Como excepción, se mantuvo la variable d_packets (diferencia entre el número acumulado de paquetes en dos instantes consecutivos dentro del mismo flujo) en el modelo de regresión, que a pesar de su alta correlación con el objetivo ($\rho = 0{,}94$), por su valor interpretativo dentro del modelo. Esta métrica representa la variación previa del tráfico y captura la inercia o dependencia temporal de los flujos, constituyendo un indicador causal legítimo. Dado que se calcula exclusivamente con información pasada, su inclusión no compromete la validez temporal del modelo ni introduce fuga de información. Su alta correlación no

[6] El coeficiente de correlación de Pearson cuantifica la relación lineal entre dos variables numéricas, tomando valores entre -1 (correlación negativa perfecta) y 1 (correlación positiva perfecta). Su aplicación en el análisis de dependencias entre métricas de red y variables objetivo es una práctica común en el aprendizaje automático aplicado a redes, según (Boutaba y cols., 2018).

refleja un problema metodológico, sino la naturaleza autoregresiva del fenómeno modelado.

El conjunto final de características incluyó:

- **Temporalidad y dinámica:** `delta_ns`, `bps`, `flow_let`, `d_packets`.
- **Control y eventos:** `loss_est_pkts`, `dup_acks_est`, flags TCP (`fin`, `syn`, `rst`, `psh`, `ack`, `urg`), `protocol`.
- **Estructura de puertos:** `src_port_b`, `dst_port_b`.

Las métricas altamente correlacionadas (*p. ej.*, *bytes*, *throughput*, *d_bytes*, *pps*) fueron excluídas por inducir dependencia directa con el objetivo o redundancia derivada del mismo fenómeno que se pretende predecir.

Cálculo de variables derivadas en el modelo regresor

Durante el preprocesamiento para el algoritmo regresor se generaron variables derivadas a partir de las métricas originales con el objetivo de capturar la dinámica temporal del tráfico dentro de cada flujo. Todas las operaciones se aplicaron respetando la causalidad temporal, es decir, utilizando únicamente información disponible en el instante actual o anterior del flujo. En particular, las derivadas se definieron de la siguiente manera:

- **Diferencia previa de paquetes (`d_packets`):** se calcula como la diferencia entre el conteo acumulado de paquetes en dos intervalos consecutivos,

$$d\_packets_t = \text{packets}_t - \text{packets}_{t-1}.$$

  Esta variable actúa como rezago temporal del objetivo (y_{t-1}), permitiendo capturar la inercia o dependencia autoregresiva del flujo.

- **Tasa de bytes por segundo (`bps`):** se obtiene como el cociente entre la variación de bytes transmitidos y el tiempo transcurrido entre muestras. Esta magnitud refleja la velocidad de transmisión efectiva en el instante t, sin involucrar información futura del flujo.

Las derivadas anteriores permiten transformar métricas acumulativas (*packets*, *bytes*) en tasas y diferencias temporales causales, adecuadas para el modelado supervisado de tráfico. De esta forma, el conjunto final de características conserva únicamente información observable en tiempo real, evitando fugas de información (*data leakage*) y garantizando la validez experimental del modelo.

3.4.2. Clasificador de congestión explícita (target CE)

Definición del objetivo. Sea CE_t el conteo (o número de paquetes) marcados con ECN-CE observado en la ventana t. El objetivo binario se define como:

$$y_t = \begin{cases} 1, & \text{si } \text{CE}_t - \text{CE}_{t-1} > 0, \\ 0, & \text{en caso contrario.} \end{cases}$$

Esto implica que el modelo aprende a *detectar la aparición de nuevos paquetes marcados con ECN-CE* (señal temprana de congestión), utilizando únicamente señales previas observables.

3.4.3. Regresor de tráfico (target: variación de *packets*)

En paralelo al clasificador, se desarrolló un modelo de regresión supervisado orientado a predecir la variación en el número de paquetes transmitidos entre dos ventanas consecutivas del mismo flujo. Esta formulación captura la dinámica del tráfico (no sus valores absolutos) y resulta útil para anticipar cambios de carga.

El objetivo se define como la diferencia entre los paquetes observados en los instantes t y $t+1$, desplazada hacia adelante por flujo:

$$y_t = \text{packets}_{t+1} - \text{packets}_t$$

Durante el preprocesamiento, el ordenamiento y los rezagos se aplicaron por flujo/flowlet para preservar causalidad y evitar fuga de información.

3.4.4. Selección del modelo XGBoost

Siguiendo el marco conceptual propuesto por Boutaba et al. (Boutaba y cols., 2018), esta sección adopta sus lineamientos generales sobre aprendizaje automático aplicado a redes. En particular, se selecciona el modelo XGBoost como base para las tareas de clasificación y regresión. XGBoost es una implementación optimizada de los *Gradient Boosted Decision Trees (GBDT)* (Chen y Guestrin, 2016), un enfoque que entrena secuencialmente árboles que corrigen los errores del conjunto previo y optimiza una función de pérdida diferenciable (por ejemplo, log-loss o Mean Squared Error (MSE) (Hastie, Tibshirani, y Friedman, 2009)). Su capacidad para capturar relaciones no lineales, su robustez frente a ruido y su eficiencia en conjuntos de datos con características mixtas lo hacen especialmente adecuado para problemas de predicción de tráfico en redes de alta capacidad.

Los modelos basados en árboles, incluidos RF y XGBoost, han mostrado en distintos estudios compilados por Boutaba et al. precisiones superiores al 90 % en tareas de clasificación de tráfico, confirmando su idoneidad para entornos de datos heterogéneos y masivos. En este trabajo, el modelo fue implementado utilizando la biblioteca `scikit-learn` (Pedregosa y cols., 2011).

3.4.5. Estrategia de entrenamiento

Para el entrenamiento y evaluación de los modelos se adoptó una estrategia de particionamiento tipo holdout con proporciones del 60 % para entrenamiento, 20 % para validación y 20 % para prueba final. Esta decisión se alinea con las buenas prácticas metodológicas descritas en la literatura especializada. Esta descomposición del dataset nos permite equilibrar la cantidad de datos dedicados al ajuste de los parámetros con una reserva suficiente para la evaluación

objetiva del rendimiento del modelo. En el contexto del análisis de tráfico de red, esta elección resulta especialmente relevante, ya que las muestras pueden presentar correlaciones temporales o espaciales entre flujos consecutivos. Por ello, se estableció además la independencia entre flujos en el proceso de división garantizando que las instancias derivadas de un mismo flujo no se distribuyan entre los distintos subconjuntos para evitar fuga de información (data leakage), preservando la validez estadística de la evaluación. En conjunto, esta estrategia proporciona una base metodológica robusta para medir la capacidad de generalización del modelo en escenarios de tráfico.

3.4.6. Métricas de evaluación de los algoritmos

La selección de métricas para evaluar los modelos propuestos se fundamenta directamente en las recomendaciones presentadas por (Boutaba y cols., 2018), quienes en su *Table 2* identifican indicadores de desempeño más utilizados en aprendizaje automático aplicado a redes. Esta tabla identifica las métricas de precisión y error que permiten evaluar la capacidad de generalización de los modelos bajo diferentes condiciones de tráfico, distribución de clases y escalas de datos.

En el modelo clasificador se emplearon las métricas resumidas en la Tabla 3.3, seleccionadas por su relevancia para tareas de clasificación binaria. Estas métricas permiten evaluar tanto la capacidad del modelo para detectar eventos de congestión como su precisión global y discriminación entre clases.

En el modelo regresor, se adoptaron métricas de error promedio y dispersión también descritas en la Tabla 3.4, siendo el *Error Absoluto Medio (Mean Absolute Error (MAE))*, *Error Cuadrático Medio (MSE)* y su raíz cuadrada, *Root Mean Squared Error (RMSE)*, además del *Coeficiente de Determinación* (R^2).

En conjunto, las métricas seleccionadas permiten una evaluación integral de los modelos de clasificación y regresión. Su elección se sustenta en las recomendaciones metodológicas presentadas en la Sección 2.5 de (Boutaba y cols., 2018), quienes señalan que la validez del aprendizaje automático en redes depende de la utilización de indicadores de desempeño sensibles al contexto de tráfico, a la escala de los datos y a la distribución de clases.

3.4.7. Interpretación de resultados y análisis de desempeño

Modelo Clasificador de Congestión Explícita

Como se mencionó anteriormente, el conjunto de datos se dividió en proporciones 60/20/20 para entrenamiento, validación y prueba respectivamente, garantizando independencia temporal y por flujo, con un total de 46535 muestras para entrenamiento, 15869 para validación y 15800 para prueba. La dis-

[7] En clasificación binaria, la *sensitivity* mide la proporción de instancias positivas correctamente identificadas, mientras que la *specificity* indica la proporción de instancias negativas correctamente clasificadas.

Tabla 3.3: Métricas empleadas para la evaluación del modelo clasificador.

Métrica	Descripción	Ref.
Precisión	Proporción de predicciones positivas que son correctas; mide la exactitud de las detecciones positivas.	(Boutaba y cols., 2018)
Recall (TPR)	Sensibilidad o tasa de verdaderos positivos; mide la capacidad del modelo para detectar correctamente las instancias positivas.	(Boutaba y cols., 2018)
F1-score	Media armónica entre *precision* y *recall*; equilibra exactitud y cobertura ante clases desbalanceadas.	(Boutaba y cols., 2018)
Accuracy	Proporción total de predicciones correctas; útil como medida global, aunque poco confiable ante clases desbalanceadas.	(Boutaba y cols., 2018)
Area Under ROC curve (AUROC)	Área bajo la curva ROC; evalúa la relación entre verdaderos y falsos positivos a distintos umbrales, reflejando *sensitivity* y *specificity*[7].	(Boutaba y cols., 2018)
Area under Precision-Recall (AUPRC)	Área bajo la curva Precisión–Recall; alternativa al AUROC, más informativa en conjuntos con clases raras o desbalanceadas.	(Saito y Rehmsmeier, 2015)
Matriz de confusión	Tabla de conteos que resume las predicciones del modelo (TP, FP, TN, FN) y permite derivar las métricas anteriores.	(Boutaba y cols., 2018)

tribución de clases fue moderadamente desbalanceada (62 % positivas y 38 % negativas). Se observa que se decidió optimizar el recall de la clase 1, maximizando la sensibilidad del sistema frente a eventos de congestión, reduciendo los falsos negativos.

En el conjunto de prueba, el Modelo Clasificador de Congestión Explícita alcanzó un rendimiento global satisfactorio, con accuracy = 0.8175, AUROC = 0.8360 y AUPRC = 0.8573, lo que evidencia una adecuada capacidad discriminativa.

Los resultados muestran un comportamiento asimétrico entre clases, consecuencia directa de la estrategia de priorización adoptada, donde el modelo fue optimizado para maximizar el recall en la clase positiva (congestión). Este enfoque va de la mano respecto a que un falso negativo (no detectar congestión real) tiene consecuencias más costosas que un falso positivo (alertar congestión inexistente)

El recall 0.939 indica que el modelo detecta correctamente más del 93 % de los eventos de congestión explícita. Por otro lado, la precisión de 0.801 muestra

Tabla 3.4: Métricas empleadas para la evaluación del modelo regresor.

Métrica	Descripción	Ref.
Mean Absolute Error (MAE)	Promedio del valor absoluto de las diferencias entre predicciones y valores reales. Evalúa la precisión media del modelo de forma directamente interpretable en las mismas unidades de la variable predicha.	(Boutaba y cols., 2018)
Mean Squared Error (MSE)	Promedio de los errores elevados al cuadrado; penaliza más fuertemente los errores grandes, lo que lo hace sensible a valores atípicos.	(Boutaba y cols., 2018)
Root Mean Squared Error (RMSE)	Raíz cuadrada del MSE; expresa la desviación estándar de los errores en las mismas unidades que la variable objetivo, reflejando la dispersión total de las predicciones.	(Boutaba y cols., 2018)
Coeficiente de determinación (R^2)	Proporción de la varianza explicada por el modelo respecto a la varianza total. Indica la capacidad global del modelo para capturar la dinámica del tráfico, siendo 1 el ajuste perfecto.	(Boutaba y cols., 2018)

Tabla 3.5: Métricas por clase del clasificador de congestión

Clase	Precision	Recall	F1-score	Soporte
0 – No congestión	0.8614	0.6183	0.7199	5 992
1 – Congestión	0.8011	0.9392	0.8647	9 808
Promedio ponderado	0.8240	0.8175	0.8098	15 800

que cerca del 20 % de las detecciones son falsas alarmas. Este equilibrio entre precisión y recall se refleja también en el F1-score de 0.8647, que resume adecuadamente la compensación entre sensibilidad y precisión.

En contraste, la clase 0 (no congestión) muestra un recall más bajo (0.618), lo que indica que aproximadamente un tercio de los casos sin congestión fueron clasificados erróneamente como positivos. Si bien esto reduce la especificad (tasa de verdaderos negativos) del modelo, es coherente con el sesgo buscado hacia la detección proactiva. El alto valor de *precision* (0.861) de esta clase demuestra, sin embargo, que las predicciones de "no congestión" son mayormente correctas cuando se emiten.

La curva ROC (AUROC = 0.836) y la curva de precision-recall (AUPRC = 0.857) confirman la buena capacidad de distinción de las clases y la estabilidad del modelo frente a distintos umbrales.

Durante los experimentos preliminares, se observó que la inclusión de cier-

tas features (throughput, delta_ns, packets, bytes y last_timestamp_ns) provocaba un incremento abrupto de las métricas de desempeño (AUROC, AUPRC y F1-score) hasta valores cercanos a 1 tanto en entrenamiento como en prueba. Aunque este resultado podría interpretarse superficialmente como un éxito del modelo, un análisis más detallado reveló que dichas variables mantienen una dependencia directa o derivada con la etiqueta de salida. En otras palabras, estas features contienen información causalmente posterior o parcialmente redundante respecto al fenómeno de congestión explícita que el modelo intenta predecir.

Por ejemplo, throughput y bytes reflejan el volumen de tráfico ya afectado por la congestión, mientras que delta_ns y last_timestamp_ns están estrechamente ligados a la dinámica temporal del flujo que define el evento Congestion Experienced (CE). Incluirlas implicaría que el modelo aprende la manifestación misma de la congestión, no su anticipación, generando una forma de data leakage temporal que invalida la interpretación predictiva de los resultados.

En consecuencia, estas variables fueron excluidas del conjunto final de entrenamiento, preservando la independencia temporal entre entradas y objetivo. Esta decisión explica que los valores de AUROC (0.836) y AUPRC (0.857) reflejen una estimación realista del poder predictivo del modelo, libre de correlaciones triviales.

En términos operativos, el modelo ofrece una alta sensibilidad y un bajo riesgo de omisión de congestión con el costo asociado de una tasa moderada de falsos positivos. El análisis detallado por clase evidencia que la estrategia de priorizar la sensibilidad fue efectiva, el sistema detecta casi todos los eventos de congestión, a costa de un número controlado de falsas alarmas.

Regresor de Tráfico

El segundo modelo se diseñó para estimar la variación del número de paquetes transmitidos entre dos ventanas consecutivas del mismo flujo, utilizando únicamente información observable en el instante actual.

En la primera versión del modelo se incluyeron todas las variables disponibles, sin control temporal explícito entre flujos ni filtrado de correlación. Los resultados iniciales (Tabla 3.6) mostraron valores aparentemente elevados de desempeño en entrenamiento y validación, pero una fuerte caída en el conjunto de prueba. Este contraste evidenció la presencia de *fuga de información (data leakage)*: variables como `bytes` y `lag_packets` presentaban correlaciones altas tanto con el número de paquetes actual como con el siguiente, es decir, contenían información del futuro inmediato del flujo. Los modelos supervisados aplicados a tráfico de red pueden incorporar inadvertidamente datos acumulativos o temporales que degradan la validez predictiva. Por ello, se procedió a depurar el conjunto de variables.

Resultando en: `delta_ns`, `loss_est_pkts`, `dup_acks_est`, `bps`, `flow_let`, las flags TCP (`fin`, `syn`, `rst`, `psh`, `ack`,`urg`), `protocol`, `src_port_b` y `dst_port_b`.

El coeficiente de determinación negativo ($R^2 = -0{,}0031$) indica que, en el conjunto de prueba, el modelo no supera la predicción trivial basada en la media.

Tabla 3.6: Resultados del modelo regresor tras depuración de variables correlacionadas

Conjunto	**MAE**	**RMSE**	R^2
Entrenamiento (TRAIN)	251.87	1,173.40	0.9176
Validación (VAL)	748.88	2,896.81	0.5277
Prueba (TEST)	1,643.30	21,854.13	−0,0031

Esto sugiere que las relaciones aprendidas no se mantienen en flujos o períodos no observados, probablemente debido a la naturaleza no estacionaria del tráfico o a la insuficiencia de las métricas seleccionadas para capturar su dinámica completa. En entrenamiento y validación, sin embargo, el modelo logra explicar buena parte de la varianza, lo que confirma que la arquitectura y el algoritmo son adecuados para aprender patrones locales estables.

La relación RMSE $\gg$ MAE con factores de diferencia más de diez demuestra la presencia de pocos errores extremadamente grandes asociados a flujos elefante que inflan la métrica cuadrática. Las Figuras 3.4 y 3.5 evidencian que el modelo predice correctamente los flujos pequeños y medianos, pero subestima los valores altos, patrón típico del tráfico con *cola pesada (heavy-tail)*. En la Figura 3.5 se utiliza una escala logarítmica en el eje Y con el fin de visualizar simultáneamente la alta concentración de errores cercanos a cero y la baja frecuencia de errores extremadamente grandes.

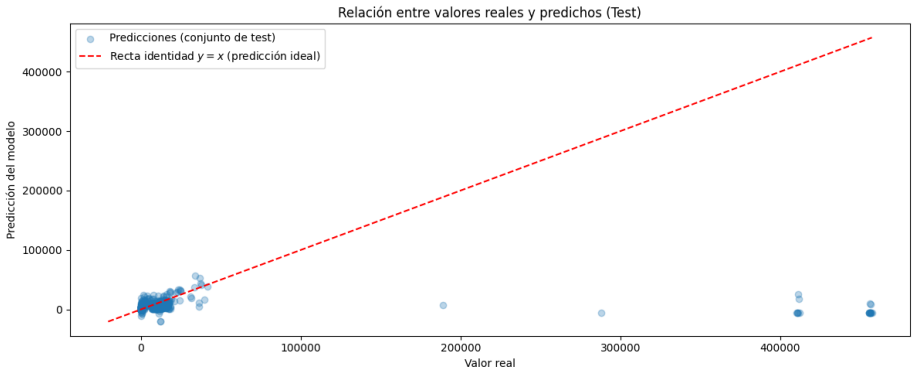

Figura 3.4: Relación entre valores reales y predichos para el target (paquetes por intervalo). La línea punteada roja indica la recta identidad $y = x$, que representa el comportamiento ideal..

Para examinar si la degradación observada en test respondía a cambios estructurales en el tráfico, se entrenaron tres modelos independientes sobre distintos períodos temporales ("early", "mid", "late") y se evaluaron recíprocamente. La matriz de desempeño R^2 resultante se muestra en la Tabla 3.7.

Se observa que cada modelo mantiene un buen desempeño cuando se evalúa dentro del mismo período en que fue entrenado, pero su capacidad predictiva

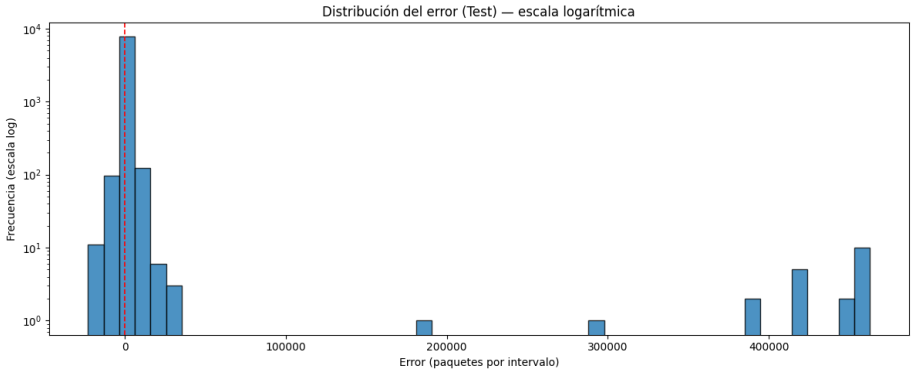

Figura 3.5: Distribución de errores en el conjunto de prueba expresado en paquetes por intervalo. Se utiliza escala logarítmica en el eje Y para visualizar tanto la alta frecuencia de errores pequeños como la baja frecuencia de errores extremadamente grandes

Tabla 3.7: Matriz de desempeño cruzado del modelo regresor según período de entrenamiento y evaluación

Entrenado/Evaluado	**early**	**mid**	**late**
early	**0.936**	0.288	0.049
mid	−0,163	**0.951**	0.059
late	−1,191	−0,496	**0.992**

se reduce considerablemente al aplicarse sobre períodos distintos. Este comportamiento indica que las relaciones entre las variables de entrada y el objetivo varían a lo largo del tiempo, probablemente debido a cambios en las condiciones de red, en la carga de tráfico o en el tipo de flujos presentes.

En términos prácticos, los resultados sugieren que la técnica empleada, id est, un modelo supervisado estático entrenado sobre un conjunto fijo de datos no es suficiente para capturar estas variaciones temporales.

Finalmente, se concluye que la caída de R^2 en el conjunto de prueba sugiere que el modelo no logra representar correctamente la nueva distribución de datos posterior al entrenamiento. En síntesis, el regresor evidencia los límites prácticos del aprendizaje supervisado sobre datos de tráfico no estacionario, donde el rendimiento disminuye ante la evolución del entorno de red y la presencia de colas pesadas (heavy-tails). No obstante, el estudio aporta evidencia valiosa para orientar futuras líneas de trabajo en reentrenamiento adaptativo y detección de anomalías, estrategias recomendadas en la literatura reciente sobre analítica de redes basada en aprendizaje automático

3.4.8. Síntesis de la etapa de modelado

La etapa de modelo tuvo como objetivo el uso de técnicas de aprendizaje supervisado para la predicción y clasificación de comportamientos de tráfico en entornos de red de alta capacidad. Se desarrollaron dos líneas complementarias:

(i) un modelo **clasificador**, orientado a la identificación de eventos de congestión a partir de métricas observables de flujo; y

(ii) un modelo **regresor**, destinado a estimar la diferencia de cantidad de paquetes transmitidos en la siguiente ventana temporal.

Ambos modelos se implementaron sobre un conjunto de datos recolectado en condiciones controladas de tráfico, con un esquema de evaluación independiente en train, validation y test.

En el caso del clasificador, se utilizó un enfoque basado en GBDT con ajuste de umbrales para optimizar la detección de la clase positiva (congestión). El modelo alcanzó un desempeño general satisfactorio, con un recall de 0.94 para la clase 1 y un F1-score de 0.86, mostrando alta sensibilidad para identificar episodios de congestión incluso a costa de un ligero aumento de falsos positivos. Esta decisión metodológica se justifica por el contexto del problema: en sistemas de control de tráfico, es preferible sobredetectar posibles congestiones antes que omitir eventos críticos. Los valores de AUROC (0.84) y AUPRC (0.86) corroboran una buena capacidad discriminativa, validando la utilidad del modelo como componente de alerta temprana dentro de una arquitectura de monitorización inteligente.

Por otro lado, el modelo regresor permitió estudiar la evolución cuantitativa del tráfico. Tras una serie de ajustes metodológicos se observó que el rendimiento del modelo descendía notablemente al pasar de entrenamiento $R^2 = 0.91$ a prueba $R^2 = -0.003$. La diferencia marcada entre *RMSE* y *MAE* (con un factor cercano a 10) evidenció la existencia de pocos flujos con errores extremadamente grandes, coherente con la naturaleza heavy-tail del tráfico de red. En tales casos, los modelos basados en boosting tienden a concentrarse en la parte densa de la distribución, perdiendo precisión frente a los denominados "flujos elefante". Esto no indica una deficiencia del dataset, sino una limitación inherente de los modelos supervisados estáticos para representar fenómenos altamente variables y asimétricos.

En conjunto, los resultados de esta etapa muestran que el aprendizaje supervisado ofrece capacidades predictivas valiosas para el monitoreo de red, pero que su eficacia depende de la estabilidad de las relaciones entre las métricas de tráfico y los fenómenos que se intentan predecir. Mientras que el clasificador demostró ser robusto ante variaciones moderadas del entorno, el regresor evidenció la sensibilidad de las aproximaciones numéricas frente a cambios de régimen o eventos anómalos. Estas observaciones respaldan la necesidad de incorporar mecanismos de actualización adaptativa, normalización por flujo o reentrenamiento periódico, de modo que los modelos mantengan su validez frente a la evolución dinámica del sistema.

3.5. Conclusiones

El desarrollo del sistema de telemetría y monitoreo predictivo implicó la toma de múltiples decisiones de diseño orientadas a maximizar el desempeño y la estabilidad bajo las limitaciones impuestas por el hardware disponible. Esta sección resume las principales elecciones arquitectónicas y las restricciones observadas durante la implementación.

3.5.1. Decisiones arquitectónicas

Se adoptó una arquitectura modular en tres niveles —SmartNIC, kernel y espacio de usuario— que permitió aislar las funciones de captura, agregación y persistencia. Los programas eBPF se diseñaron con lógica mínima, dedicándose únicamente a la clasificación y conteo, mientras que las tareas de procesamiento intensivo se trasladaron al espacio de usuario.

El despliegue en la SmartNIC Netronome se realizó en modo *offload* parcial, manteniendo en el kernel las funciones de sincronización y gestión de mapas (NFP). Esta estrategia equilibró rendimiento y compatibilidad con las restricciones del verificador del kernel y las limitaciones de memoria del dispositivo.

Los mapas eBPF se mantuvieron *pinneados* en `/sys/fs/bpf/`, garantizando persistencia y acceso concurrente. Esta decisión simplificó la comunicación entre niveles y posibilitó la ejecución simultánea de múltiples procesos de monitoreo sin reinyección del código eBPF.

La recolección de métricas del sistema mediante `get_sys_metrics.c` se integró con el flujo de red para proveer contexto adicional al análisis. Para el almacenamiento se emplearon consultas SQL directas con la biblioteca `libpq`, evitando el uso de Object-Relational Mapping (ORM) y garantizando un rendimiento óptimo en la inserción de datos en TimescaleDB.

3.5.2. Limitaciones observadas

Durante la implementación se identificaron restricciones asociadas principalmente al hardware Netronome NFP-4000 y al ecosistema de herramientas eBPF:

- **Memoria limitada en la SmartNIC:** los mapas eBPF poseen un tamaño máximo fijo, restringiendo la cantidad de información almacenable.
- **Restricciones del verificador:** el verificador de la Netronome impone condiciones más estrictas que el del kernel Linux, limitando el uso de bucles o estructuras dinámicas.
- **Imposibilidad en las llamadas a funciones:** operaciones como `bpf_map_update_elem()` no es posible utilizarlas en modo *offload*, lo que obligó a simplificar la lógica de actualización.

Estas observaciones fueron consideradas para el diseño de los experimentos y se documentan en detalle en el capítulo correspondiente a la **Experimentación**.

Capítulo 4

Experimentación

Este capítulo describe el proceso experimental realizado para evaluar el funcionamiento del sistema de telemetría programable y la efectividad de los modelos predictivos desarrollados. Se detalla la configuración del entorno, la metodología de pruebas, los resultados obtenidos y el análisis de las métricas de desempeño tanto del sistema como de los modelos de aprendizaje supervisado.

4.1. Configuración del entorno de pruebas

Las pruebas se realizaron en un único equipo del proyecto de investigación SmartLAB de la Facultad de Ingeniería, equipado con una tarjeta **SmartNIC Netronome Agilio CX NFP-4000**. Dicha tarjeta dispone de dos interfaces físicas(IF_0, IF_1), conectadas mediante un cable de FO.

A nivel de software se configuró un namespace ("namespaces(7) — Linux manual page", 2024), lo cual es un mecanismo de aislamiento que permite que distintos procesos vean y utilicen recursos del sistema de forma independiente. Cada namespace mantiene su propia pila TCP/IP e interfaces de red virtuales, lo que posibilita crear entornos de red completamente aislados dentro del mismo host (Kerrisk, 2012). En este esquema, la interfaz IF_0 se movió al namespace **ns1**, mientras que la interfaz IF_1 permaneció en el namespace **default**. Esta configuración permitió simular el tránsito de tráfico entre un nodo emisor y un nodo receptor dentro del mismo host, manteniendo aislamiento entre ambos entornos de red.

La Figura 4.1 muestra la disposición de los *network namespaces* utilizados. Sobre una de las interfaces se enganchó el programa XDP, encargado de interceptar y procesar los paquetes en el plano de datos.

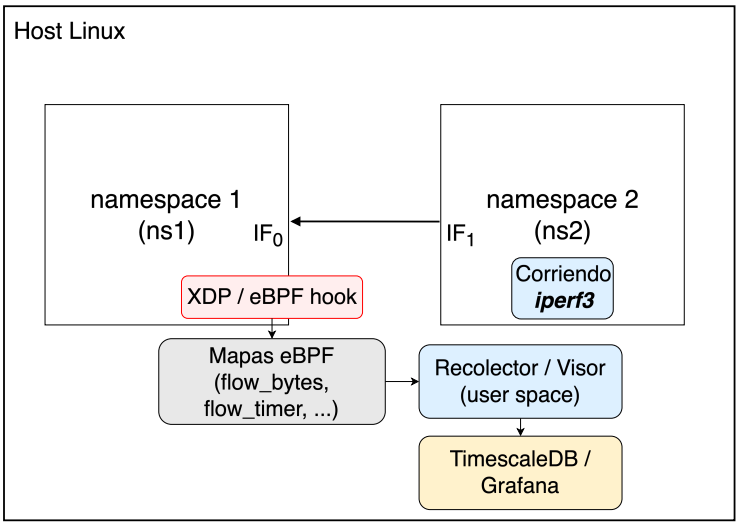

Figura 4.1: Entorno experimental utilizado para la validación del sistema de monitoreo. El tráfico TCP generado por `iperf3` dentro del *namespace* `ns2` atraviesa las interfaces virtuales IF_1–IF_0, donde es interceptado por el programa eBPF/XDP. Las métricas recolectadas se almacenan en mapas eBPF y son posteriormente consultadas por el recolector en espacio de usuario para su registro en TimescaleDB y visualización en Grafana.

El tráfico se generó con la herramienta **iperf3**, utilizando las dos interfaces de la SmartNIC: una como origen (IF_0) y la otra como destino (IF_1) del flujo. De este modo, los paquetes atravesaban el cable, pasando por el programa eBPF/XDP en modo *offload*, que se ejecutaba sobre la interfaz de recepción. Esta configuración permitió realizar las mediciones, congestión y carga de CPU sin requerir múltiples equipos físicos.

Los datos recolectados se almacenaron en **TimescaleDB** y se visualizaron mediante **Grafana**.

Los principales parámetros experimentales se resumen en la Tabla 4.1.

4.2. Casos de prueba

El conjunto de programas desarrollado permitió reproducir condiciones controladas de congestión en un entorno experimental, con el propósito de validar la funcionalidad del sistema y comparar el rendimiento del programa eBPF ejecutado en modo driver frente al modo offload (SmartNIC).

Los casos de prueba fueron diseñados para:

1. Generar congestión en la transmisión mediante la variación periódica del ancho de banda así como otras medidas disponibles.

Tabla 4.1: Parámetros generales del entorno experimental.

Parámetro	**Valor**
Kernel Linux	6.4 con soporte eBPF/XDP
Tarjeta de red	SmartNIC Netronome Agilio CX NFP-4000
Interfaces utilizadas	NFP-6xxx ver 0.0.3.5
Herramienta de tráfico	`iperf3`
Duración del experimento	20 minutos
Intervalo de muestreo	0.1 segundos
Base de datos de métricas	PostgreSQL 14.17 con TimescaleDB 2.19.3
Visualización	Grafana v12.2.0

2. Capturar métricas TCP de cada conexión con una frecuencia de muestreo elevada.

3. Producir datos para el entrenamiento de un modelo predictivo de ML.

El sistema de ejecución de pruebas se estructuró en un componente principal u orquestador de escenarios programado en Bash. Este orquestador es responsable de configurar la cola de transmisión, establecer las tasas máximas y mínimas, lanzar las instancias de `iperf3`, iniciar la captura de métricas y realizar la limpieza del entorno al finalizar cada ejecución. Se empleó la combinación `HTB + fq_codel ecn` para regular el flujo de salida en la interfaz seleccionada, implementando así un cuello de botella artificial controlable.

Para inducir fases alternadas de congestión y no congestión, se configuró la disciplina `HTB` como limitador de tasa (clase `1:1`) alternando entre dos valores de referencia: `rate_high` y `rate_low`. Sobre la clase se acopló la cola `fq_codel ecn` con los parámetros:

- `target = 5ms`, que define el retardo objetivo por paquete;
- `interval = 100ms`, que establece el período de control de la cola;
- `ce_threshold = 4ms`, cuando es soportado por el kernel.

Se habilitó el uso de ECN tanto en el cliente como en el servidor mediante el parámetro `net.ipv4.tcp_ecn=2`. De esta forma las colas estaban habilitadas a marcar los paquetes con la flag ECN Capable y por tanto están en todas las capacidades de marcar los paquetes en el caso de que se cumplan las condiciones.

Durante la ejecución, la tasa se modificó de manera oscilante entre los valores alto y bajo con variabilidad controlada en cada ciclo. Para cada período, se seleccionaron aleatoriamente:

- el valor del período total entre $[P_{\text{mín}}, P_{\text{máx}}]$;
- el porcentaje de tiempo en tasa baja (D_{low}) entre $[D_{\text{mín}}, D_{\text{máx}}]$;

- una perturbación de jitter independiente de hasta $\pm J\,\%$ sobre ambos niveles de tasa.

De esta manera, cada ciclo presentó características levemente diferentes, permitiendo observar respuestas diversas del mecanismo de control de congestión manteniendo un contexto de variabilidad realista.

Durante las pruebas se aplicaron los siguientes valores base:

- **Interfaz:** IF_0 (emisor).
- **Flujos:** 1 proceso `iperf3` con 32 conexiones paralelas (`-P 32`).
- **Separación entre inicios:** 0,2 segundos.
- **Intervalo de muestreo:** 0,01 segundos (100 Hz).
- **Período entre `rate_high` y `rate_low`:** entre 8 y 18 segundos.
- **Proporción en `rate_low`:** entre 30 % y 50 % del período.
- **Jitter:** hasta ±5 % sobre las tasas configuradas.

4.2.1. Escenarios experimentales

El diseño de los escenarios experimentales buscó reproducir distintas condiciones de carga que permitieran evaluar la sensibilidad del sistema de monitoreo y de los modelos predictivos ante variaciones abruptas y no estacionarias del tráfico. En redes de alta capacidad, los patrones de congestión suelen surgir a partir de oscilaciones rápidas en la tasa de envío y alternancias periódicas entre estados de alta y baja utilización, fenómenos ampliamente documentados en estudios de tráfico en datacenters (Alizadeh, Greenberg, Maltz, y cols., 2010; Roy, Zeng, Bagga, Porter, y Snoeren, 2015; Kandula, Sengupta, Greenberg, Patel, y Chaiken, 2009). . Cada escenario tuvo una duración de 120 segundos y fue precedido y sucedido por un período de estabilización de 5 segundos.

Tabla 4.2: Escenarios de congestión con variabilidad por ciclo.

ID	Etiqueta	Duración (s)	rate_{high}	rate_{low}	Rango duty/periodo (%)
301	Ultra-agresivo	120	300 Mbit	20 Mbit	60–80, 4–10 s, jitter ±2 %
302	Agresivo 2	120	200 Mbit	40 Mbit	50–70, 6–12 s, jitter ±3 %
303	Ráfagas	120	180 Mbit	60 Mbit	40–60, 3–8 s, jitter ±1 %
304	Extremo	120	400 Mbit	10 Mbit	70–90, 5–10 s, jitter ±0 %
201	Moderado	120	200 Mbit	60 Mbit	30–50, 8–18 s, jitter ±5 %
202	Agresivo	120	150 Mbit	30 Mbit	30–50, 10–20 s, jitter ±7 %
203	Suave	120	250 Mbit	100 Mbit	30–50, 8–16 s, jitter ±3 %

Entre escenarios consecutivos se estableció una pausa de 35 segundos con el fin de evitar interferencias temporales entre ejecuciones.

El procedimiento seguido en cada ejecución fue el siguiente:

1. Se habilitó el soporte ECN en el cliente y el servidor.
2. Se configuró la cola de control `htb + fq_codel ecn` sobre la interfaz de salida.
3. Se iniciaron las cargas `iperf3`.
4. Se mantuvo la alternancia de tasas según el plan durante 120 segundos.
5. Se detuvo el muestreo una vez finalizadas todas las conexiones y se eliminó la configuración de colas.
6. Se esperó un período de 35 segundos antes de iniciar el escenario siguiente.

4.2.2. Consideraciones de reproducibilidad y validez

Se implementaron varios mecanismos para asegurar la reproducibilidad de las pruebas:

- La dirección IP de la interfaz utilizada se fijó explícitamente en las cargas `iperf3` para garantizar el paso del tráfico a través del cuello de botella configurado.
- La limpieza del entorno fue completa, eliminándose siempre la configuración de colas antes de finalizar cada escenario.

El conjunto de scripts implementado permitió generar de manera controlada episodios de congestión con diferentes grados de intensidad y variabilidad temporal. El esquema configurado mediante `htb + fq_codel ecn` resultó adecuado para forzar transiciones entre estados de congestión y descarga de cola. El programa receptor de métricas posibilitó la recolección de información de transporte a alta frecuencia, obteniéndose trazas con precisión temporal suficiente para alimentar los modelos de aprendizaje automático desarrollados en etapas posteriores.

4.3. Resultados de recolección y desempeño del sistema

Durante la ejecución de los experimentos, el sistema mantuvo una tasa estable de recolección de métricas con una granularidad de 0.1 segundos, sin generar saturación apreciable en el CPU principal.

Las métricas obtenidas desde los mapas eBPF se integraron correctamente con las estadísticas del sistema operativo, lo que permitió correlacionar variaciones en el `rtt` con cambios en la utilización de CPU y memoria.

El conjunto completo de métricas recolectadas y derivadas se presenta en el Anexo A.1.

Visualización

La visualización de resultados se realizó mediante paneles en Grafana, donde fue posible analizar la evolución temporal de los flujos de red y distintos indicadores del sistema. La Figura 4.2 muestra un ejemplo del panel utilizado para monitorizar el histórico de flujos, la tasa de paquetes procesados y otras métricas relevantes.

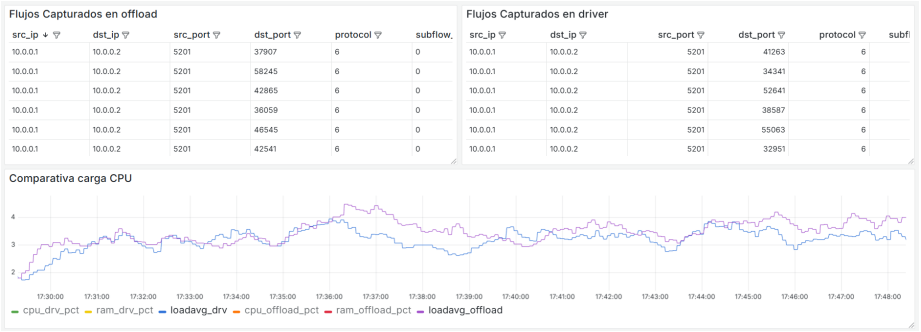

Figura 4.2: Dashboard de Grafana mostrando histórico de flujos y métricas.

Las métricas recolectadas contemplan muchos aspectos del uso de CPU y RAM, por lo tanto se toma la decisión de hacer algunos cálculos para llegar a una medida contemplativa de las métricas recolectadas pero mas resumida, es decir %CPU y %RAM. La explicación de los cálculos hechos para llegar a estas nociones están disponibles en el Anexo B

4.3.1. Análisis

En esta sección se presentan los resultados obtenidos al ejecutar el programa eBPF en los modos *driver* y *offload*, con el objetivo de evaluar el impacto de cada modalidad sobre el rendimiento del procesamiento de paquetes y el consumo de recursos del sistema.

Las métricas analizadas incluyen la tasa de procesamiento (*packets per second*, PPS), el porcentaje de utilización de CPU, la cantidad de tiempo en ticks (x100) del CPU, el porcentaje de memoria, el throughput y la variabilidad temporal de dichas medidas.

Tabla 4.4: Paquetes por segundo en promedio y desviación estándar por modo de ejecución.

Modo	**PPS medio**	**Desviación estándar (PPS)**
Driver	4369	13136
Offload	12656	83002

La Figura 4.4 muestra la evolución temporal del número de paquetes procesados por segundo en ambos modos. En el modo *offload*, se observa una mayor

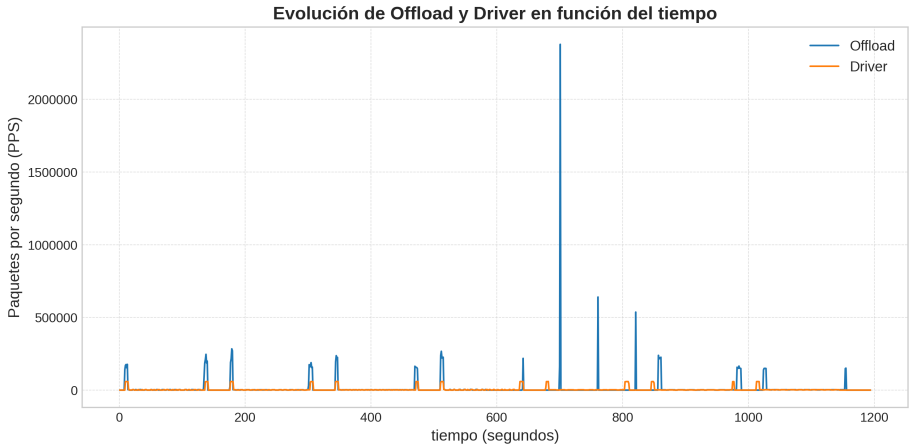

Figura 4.3: Comparación de la tasa de procesamiento (PPS) entre los modos *driver* y *offload*.

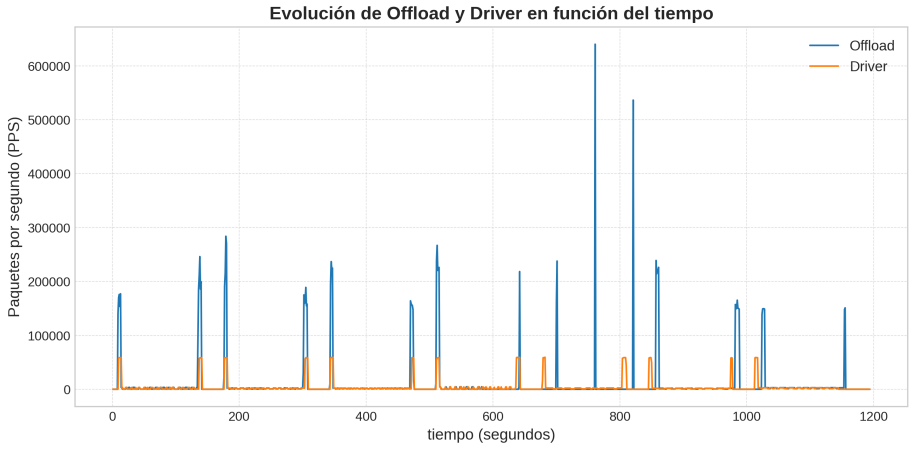

Figura 4.4: Comparación de PPS modificada para mejor visualización.

tasa de procesamiento promedio, asociada a la ejecución directa en la SmartNIC. Por el contrario, el modo *driver* presenta una tasa ligeramente inferior pero más estable a lo largo del tiempo.

De acuerdo a la forma en que fueron planteados los casos de prueba podemos ver los picos de inicio de cada una de las corridas de los casos. En estos primeros y últimos 5 segundos de medidas el filtro asociado al caso de prueba aun no se encontraba cargado, de esta manera se obtiene una división visual de inicio y fin de cada caso. Así vemos también el espaciado temporal entre casos de 35 segundos.

A partir de los resultados de la Tabla 4.4, se observa que el modo *offload* alcanzó una tasa de procesamiento promedio aproximadamente un 189.5 % superior respecto al modo *driver*. No obstante, también presentó una mayor variabilidad, evidenciada por la desviación estándar sextuplicada, lo que podría indicar una estabilidad temporal menor en el flujo de procesamiento.

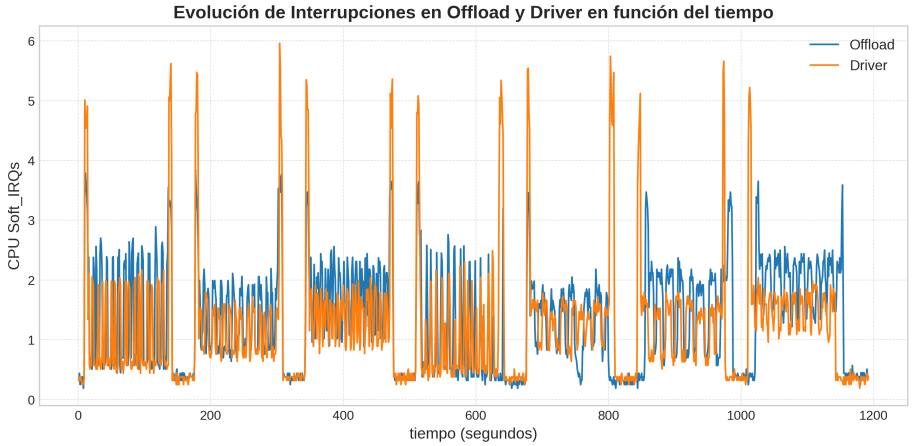

Figura 4.5: Tiempo consumido atendiendo soft irq en los modos *driver* y *offload*.

Tabla 4.5: Rendimiento promedio y desviación estándar por modo de ejecución (CPU).

Modo	**%CPU medio**	**Desviación estándar (%CPU)**
Driver	19.6	3.34
Offload	22.6	5.25

La Figura 4.5 representa la evolución temporal del consumo de CPU durante el procesamiento de tráfico. En modo *driver*, la carga de CPU se concentró en los núcleos asociados al stack de red, alcanzando valores del orden de 19 %. En modo *offload*, la utilización del CPU principal aumentó a valores promedio de 22 %. Lo que vemos es un aumento de interrupciones (*softirq*) generadas por la

sincronización de los mapas eBPF entre el host y la SmartNIC. Este fenómeno explica la mayor variabilidad observada en la tasa de procesamiento.

Tabla 4.6: Rendimiento promedio y desviación estándar por modo de ejecución (Random Access Memory (RAM)).

Modo	%RAM medio	Desviación estándar (%RAM)
Driver	31.0	0.389
Offload	30.6	0.293

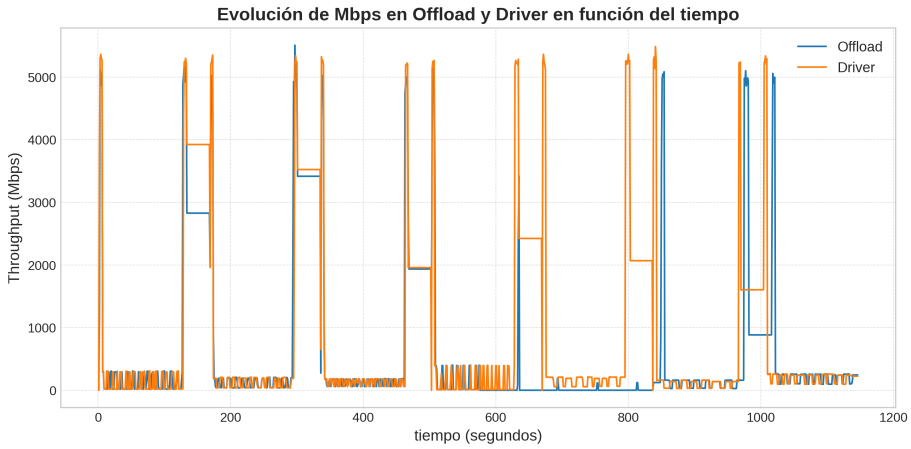

Figura 4.6: Throughput en los modos *driver*(amarillo) y *offload*(verde).

Por la Figura 4.6 y la Tabla 4.7 podemos hacer un análisis comparativo de throughput. La evidencia indica que la modalidad driver logra un caudal promedio superior, pero con mayor dispersión temporal. La alternativa offload presenta un throughput más estable, aunque ligeramente inferior en promedio, alcanzando valores máximos equivalentes. Esto sugiere que, si bien el offload no aumenta el caudal pico, podría aportar mayor regularidad y previsibilidad bajo carga sostenida, mitigando la variabilidad inducida por el procesamiento en CPU.

Tabla 4.7: Throughput promedio y desviación estándar por modo de ejecución.

Modo	Throughput medio	Desviación estándar
Driver	857 Mbps	1448 Mbps
Offload	588 Mbps	1222 Mbps

Por otro lado, el uso de memoria RAM es bastante parejo en ambos modos. Como indica la Tabla 4.6 no hay variaciones significativas.

En síntesis, se verificó que el modo *offload* permite alcanzar mayores tasas de procesamiento de paquetes, aunque introduce una variabilidad superior, menor throughput y un incremento en la frecuencia de interrupciones del sistema. El modo *driver* mostró un rendimiento más estable, con menor dispersión en las métricas, mejor throughput y mejor control sobre el uso de CPU. Estas observaciones evidencian la necesidad de optimizar la sincronización de los mapas eBPF cuando se ejecutan en la SmartNIC, a fin de reducir la sobrecarga de comunicación con el host.

4.3.2. Análisis cualitativo

El análisis cualitativo complementó la evaluación cuantitativa de rendimiento, permitiendo interpretar las diferencias observadas entre los modos de ejecución *driver* y *offload*. A partir de los registros obtenidos se identificaron patrones recurrentes en la utilización de CPU, throughput, la frecuencia de interrupciones y la estabilidad temporal del flujo de paquetes procesados.

En primer lugar, se observó que el modo *offload* alcanzó una mayor velocidad de recepción con un promedio de throughput más bajo pero más estable, lo que confirma que la descarga de procesamiento hacia la SmartNIC reduce significativamente la intervención del sistema operativo principal. Sin embargo, dicha ganancia en pps vino acompañada de una mayor variabilidad y un incremento en la tasa de interrupciones (*softirq*) detectadas en el host. Esta situación sugiere que, aunque el procesamiento de paquetes ocurre físicamente en la SmartNIC, el intercambio continuo de información con el sistema huésped introduce un costo adicional en términos de sincronización y comunicación.

El incremento en el consumo de CPU observado en el host no se asoció a limitaciones de concurrencia dentro de la SmartNIC, ya que el programa eBPF se ejecutó de forma paralela sobre múltiples *micro-engines* de la tarjeta Netronome, cada uno con varios hilos cooperativos de ejecución (Miano y cols., 2024; Netronome Systems, Inc., 2018). En este contexto, los mapas se mantuvieron en modo de solo lectura desde el espacio de usuario. El origen de la sobrecarga se atribuye, en cambio, al mecanismo de sondeo periódico de los mapas eBPF desde el host: cada consulta desencadena una transición entre el espacio de usuario y el kernel y, en el modo *offload*, una transacción adicional a través del bus PCIe para acceder al contenido de los mapas residentes en la tarjeta. Este proceso genera un número considerable de interrupciones suaves (*softirq*) y un incremento acumulativo del uso de CPU, especialmente cuando la frecuencia de muestreo es elevada. Por consiguiente, la carga observada no proviene del procesamiento de paquetes en la SmartNIC —que se realiza de manera concurrente en múltiples núcleos—, sino de la comunicación continua entre el plano de control del host y el plano de datos ejecutado en el dispositivo.

A partir de estos resultados, se identificaron varias estrategias de mejora orientadas a incrementar la eficiencia del modo *offload*. Entre ellas se destacan:

- la implementación de mecanismos de procesamiento funcional directamente en la SmartNIC, reduciendo la necesidad de sincronización con el host;

- la disminución de la frecuencia de actualización de los mapas mediante políticas de agregación temporal o muestreo adaptativo;

En términos cualitativos, el modo *offload* demostró un potencial superior en capacidad de procesamiento bruto, pero también una sensibilidad mayor a los mecanismos de comunicación entre la SmartNIC y el host. El análisis evidenció que la eficiencia total no depende únicamente de la potencia de cómputo disponible en la tarjeta, sino de la forma en que se administran los intercambios de estado y la coherencia de los mapas eBPF distribuidos. Optimizar dichos mecanismos constituye, por tanto, una línea de trabajo prioritaria para maximizar el aprovechamiento del entorno de ejecución *offload*.

4.4. Conclusiones

El conjunto de experimentos realizados permitió evaluar el comportamiento del sistema de monitoreo en los modos de ejecución *driver* y *offload*, analizando tanto el rendimiento del procesamiento de paquetes como la carga generada sobre el procesador principal. Los resultados obtenidos mostraron que la ejecución en la SmartNIC proporcionó una mayor tasa de procesamiento de paquetes (pero menor *throughput*) respecto del modo *driver*, confirmando la efectividad del desplazamiento parcial del plano de datos hacia el dispositivo de red.

No obstante, también se observó un incremento en la variabilidad temporal de las métricas y un aumento en la frecuencia de interrupciones (*softirq*) asociadas a la comunicación entre el host y la SmartNIC. Dicho fenómeno se vinculó con la frecuencia de lectura de los mapas eBPF desde el espacio de usuario y las transacciones PCIe necesarias para acceder a los datos almacenados en la tarjeta. En contraposición, el modo *driver* presentó un comportamiento más estable y un uso de CPU más uniforme, resultando más adecuado para ejecuciones prolongadas o escenarios de observación continua.

En términos generales, se concluye que la SmartNIC ofrece ventajas significativas en rendimiento bruto, aunque su aprovechamiento óptimo depende de la eficiencia con la que se gestionen las operaciones de lectura y sincronización entre ambos planos. El modo *offload* requiere, por tanto, ajustes en la frecuencia de muestreo y en la estrategia de acceso a los mapas eBPF, a fin de reducir la sobrecarga de interrupciones y mejorar la estabilidad del sistema.

En el plano analítico, los modelos supervisados basados en XGBoost presentaron resultados diferenciados.

El clasificador de eventos ECN-CE alcanzó métricas sólidas (Accuracy = 0.82, AUROC = 0.84, AUPRC = 0.86), con un recall de 0.94 para la clase positiva, validando su utilidad como detector proactivo de congestión. Las características más influyentes fueron `loss_est_pkts` y `dup_acks_est`, junto con flags TCP y medidas temporales.

En contraste, el regresor para la variación de paquetes mostró una fuerte pérdida de generalización ($R^2_{train} = 0{,}92$ frente a $R^2_{test} \approx 0$), reflejo de la dificultad de capturar dinámicas no estacionarias mediante modelos estáticos. La

diferencia entre *RMSE* y *MAE* (factor >10) revela la presencia de pocos errores muy grandes, asociados a flujos *elefante*, lo que coincide con la naturaleza *heavy-tail* del tráfico. Este comportamiento subraya la necesidad de modelos adaptativos o basados en segmentación de flujos para mejorar la capacidad de predicción en escenarios heterogéneos.

La discusión general confirma la validez de la arquitectura propuesta: la integración entre telemetría programable (eBPF/XDP), persistencia temporal y modelos supervisados constituye un enfoque viable para el monitoreo proactivo de congestión en redes de alta capacidad. Las limitaciones detectadas —como memoria restringida para mapas, tope de instrucciones del verificador y ausencia de mapas por núcleo— fueron atenuadas mediante políticas de *offload* parcial y sincronización periódica entre la SmartNIC y el host.

Capítulo 5

Conclusiones y Trabajo Futuro

Este capítulo presenta las conclusiones generales del proyecto y las líneas de trabajo que se consideran más prometedoras para su continuación. Se discuten los principales aportes alcanzados, las limitaciones encontradas y las perspectivas de evolución hacia un sistema de monitorización predictiva más completo y adaptable.

5.1. Conclusiones generales

El proyecto logró diseñar, implementar y validar una arquitectura funcional para la telemetría inteligente en redes de alta capacidad, combinando procesamiento en la SmartNIC, recolección en el kernel y analítica en el espacio de usuario. La integración de estas capas permitió construir un flujo de información continuo, eficiente y con granularidad temporal suficiente para el entrenamiento de modelos de aprendizaje automático.

Entre los principales logros alcanzados se destacan:

- La implementación de programas eBPF/XDP en modo *offload* sobre la tarjeta **Netronome Agilio CX NFP-4000**, capaces de capturar, clasificar y contabilizar tráfico de red en tiempo real.
- El diseño de una infraestructura modular y escalable que integra componentes de captura, procesamiento, persistencia y visualización del tráfico, utilizando herramientas abiertas como **TimescaleDB** y **Grafana**.
- La generación de un conjunto de datos coherente y etiquetado, representativo de diferentes condiciones de congestión, a partir del cual se entrenaron modelos supervisados con resultados satisfactorios.

- La validación de que métricas derivadas directamente de la telemetría programable —como la pérdida de paquetes y los ACKs duplicados— constituyen predictores robustos de congestión en tráfico TCP.

Los resultados experimentales demostraron que la telemetría basada en eBPF y SmartNICs puede alimentar de manera efectiva modelos de predicción de congestión, reduciendo la dependencia de instrumentaciones externas o de alto costo computacional. Asimismo, la arquitectura propuesta evidenció una buena estabilidad operativa, manteniendo tasas constantes de captura y procesamiento bajo distintas condiciones de carga.

5.2. Limitaciones del sistema

Si bien el sistema alcanzó los objetivos planteados, se identificaron varias limitaciones que condicionan su escalabilidad y generalización a entornos de producción:

- **Restricciones de hardware:** la SmartNIC Netronome NFP-4000 posee una memoria limitada, lo que restringe la cantidad de información que puede almacenarse. Por entrada en un mapa eBPF se permite un máximo de 64 bytes divididas como *key+value* y en general se permite hasta 3 000 000 de entradas. (Miano y cols., 2024).
- **Verificador restrictivo:** las reglas de validación del compilador interno de la Netronome impiden el uso de bucles dinámicos y estructuras de control complejas, limitando el grado de expresividad de los programas.
- **Ausencia de mapas por núcleo:** la tarjeta no soporta *per-CPU maps*, de modo que todas las instancias de los *micro-engines* acceden a un mismo espacio de memoria compartido. Esta característica puede generar problemas de concurrencia por lo tanto se obliga a usar operaciones atómicas o bloqueos (*spinlocks*) que reducen el rendimiento (Netronome Systems, Inc., 2018; Miano y cols., 2024).
- **Limitación en el tamaño de los programas:** el firmware de la Netronome restringe el tamaño de los programas offloadeados a unas 8 000 instrucciones, lo que obliga a adoptar una estrategia de *offload* parcial y mantener parte de la lógica en el host (Miano y cols., 2024).
- **Escalabilidad del almacenamiento:** aunque TimescaleDB mostró un buen rendimiento en volúmenes moderados, la inserción simultánea de métricas de alta frecuencia podría requerir técnicas adicionales de particionamiento o agregación.
- **Dependencia del pipeline de usuario:** parte de la lógica de limpieza y consolidación de datos se ejecuta aún en el espacio de usuario, lo que introduce latencias adicionales.

- **Compatibilidad limitada:** el soporte de eBPF offload varía entre versiones de kernel y firmware; algunas funcionalidades recientes (como *bpf timers* o *tail calls* en cascada) aún no están disponibles en el hardware Netronome, reduciendo las posibilidades de extender el sistema a nuevos casos de uso (Miano y cols., 2024).

Estas limitaciones fueron gestionadas mediante estrategias de *offload* parcial, pinning de mapas e integración modular, pero siguen representando un desafío relevante para futuras versiones del sistema.

El trabajo de (Miano y cols., 2024) optó por evitar el uso de eBPF maps para la comunicación entre el programa offloadeado y el espacio de usuario, debido a que la tarjeta Netronome no soporta maps por núcleo. Dado que la SmartNIC posee decenas de Micro-Engines que ejecutan hilos concurrentes, la utilización de maps globales podría generar contención y problemas de sincronización, afectando el rendimiento. En su lugar, los autores decidieron emplear la vía de datos (data path) para reenviar las tuplas procesadas como paquetes normales, obteniendo mayor desempeño y aislamiento entre el procesamiento en hardware y en software.

5.3. Líneas de trabajo futuro

A partir de las experiencias obtenidas, se proponen varias direcciones de trabajo futuro, tanto a nivel de infraestructura como de analítica avanzada:

- **Evolución del plano de datos.** Extender las capacidades del plano de datos mediante el uso de mapas per-CPU, que permitirían reducir la contención en actualizaciones concurrentes. Asimismo, incorporar *tracepoints*, *perf events* o sondas *kprobe/uprobes* para recolectar métricas internas del kernel que complementen las del tráfico de red. Otra mejora potencial es explorar nuevas arquitecturas de SmartNICs con soporte extendido para eBPF y P4, que ofrezcan mayor memoria y compatibilidad con bibliotecas modernas de interacción con el *datapath*.

- **Aprendizaje automático y predicción en línea.** Implementar aprendizaje en línea (*online learning*) permitiría que los modelos ajusten sus parámetros de forma incremental a medida que se reciben nuevos datos, adaptándose dinámicamente a cambios en el tráfico y el comportamiento de la red, y reduciendo la necesidad de reentrenamientos periódicos. Integrar técnicas de aprendizaje no supervisado contribuiría a la detección automática de patrones anómalos sin requerir etiquetas previas, complementando la clasificación binaria actual. A futuro, las predicciones de congestión podrían integrarse con mecanismos de gestión de colas y controladores SDN, habilitando decisiones proactivas sobre tasas de envío y balanceo de carga.

- **Integración y despliegue en entornos reales.** Avanzar hacia la integración del sistema en escenarios de red más complejos, como topologías de

data centers o infraestructuras virtualizadas. La incorporación de herramientas de orquestación (por ejemplo, `Kubernetes`) permitiría desplegar el sistema de telemetría de manera distribuida, con múltiples agentes recolectores coordinados por un *backend* central. Además, una integración más directa con **Grafana** reduciría los procesos intermedios de exportación y habilitaría monitoreo visual en tiempo real.

5.4. Aplicaciones del sistema de monitoreo y predicción

El sistema desarrollado combina capacidades de monitoreo a nivel de flujo con mecanismos de predicción basados en aprendizaje automático, ofreciendo un conjunto de funcionalidades aplicables tanto en entornos de investigación como en infraestructuras de producción. Su arquitectura modular y el uso de tecnologías estándar (eBPF, SmartNICs, PostgreSQL/TimescaleDB y modelos de *ML*) permiten su adaptación a distintos escenarios de red.

5.5. Reflexión final

El trabajo realizado evidencia el potencial de combinar la telemetría programable con el aprendizaje automático para crear sistemas de monitoreo proactivos, capaces de anticipar condiciones de congestión y degradación del rendimiento antes de que afecten a los usuarios.

Más allá de los resultados técnicos obtenidos, el proyecto contribuye a la comprensión y validación experimental de arquitecturas de telemetría distribuida sobre SmartNICs, un campo de investigación en expansión que combina ingeniería de redes, sistemas operativos y ciencia de datos.

Las bases conceptuales y prácticas establecidas en este trabajo permiten proyectar futuros desarrollos orientados a redes más inteligentes, adaptativas y sostenibles, donde la observabilidad y la predicción se integren como componentes esenciales del ciclo de gestión de rendimiento.

Referencias

Aitken, P., Claise, B., y Trammell, B. (2013, septiembre). *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information* (n.º 7011). RFC 7011. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc7011 doi: https://doi.org/10.17487/RFC7011

Alizadeh, M., Greenberg, A., Maltz, D. A., y cols. (2010). Data center TCP (DCTCP). En *Proc. acm sigcomm.*

Aramide, O. (2024, 04). Programmable Data Planes (P4, eBPF) for High-Performance Networking: Architectures and Optimizations for AI/ML Workloads. *SAMRIDDHI A Journal of Physical Sciences Engineering and Technology*, *16*, 108-117. doi: https://doi.org/10.18090/samriddhi.v16i02.07

Barroso, L., Clidaras, J., y Hölzle, U. (2013). *The datacenter as a computer: An introduction to the design of warehouse-scale machines.* Morgan & Claypool.

Beyer, B., Jones, C., Petoff, J., y Murphy, N. R. (2016). *Site reliability engineering.* O'Reilly Media.

Blog, C. (2022). *How we used eBPF to build programmable packet filtering in magic firewall.* https://blog.cloudflare.com/programmable-packet-filtering-with-magic-firewall/.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., ... Walker, D. (2014, julio). P4: programming protocol-independent packet processors. En (Vol. 44, p. 87–95). New York, NY, USA: Association for Computing Machinery. doi: https://doi.org/10.1145/2656877.2656890

Boutaba, R., Salahuddin, M. A., Limam, N., Ayoubi, S., Shahriar, N., Estrada-Solano, F., y Caicedo, O. M. (2018). A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, *9*(1), 16. doi: https://doi.org/10.1186/s13174-018-0087-2

Brandino, B., y Grampín, E. (2024). Network data plane programming languages: A survey. *Computers*, *13*(12). doi: https://doi.org/10.3390/computers13120314

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32. doi: https://doi.org/10.1023/A:1010933404324

Chauhan, M. (2025). *Meta's strobelight uses eBPF to boost efficiency and cut server costs.* Descargado de https://tfir.io/metas-strobelight-uses-ebpf-to-boost-efficiency-and-cut-server-costs

Chen, T., y Guestrin, C. (2016). XGBoost: A scalable tree boosting system. En *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (kdd '16)* (pp. 785–794). New York, NY: ACM. doi: https://doi.org/10.1145/2939672.2939785

Cilium Project. (2025). Cilium documentation: eBPF component overview [Manual de software informático]. Descargado de https://docs.cilium.io/en/stable/overview/component-overview/#ebpf

Cisco Systems. (2020). *Cisco annual internet report (2018–2023) white paper.* Descargado de https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

Claise, B. (2004, octubre). *Cisco Systems NetFlow Services Export Version 9* (n.º 3954). RFC 3954. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc3954 doi: 10.17487/RFC3954

Cole, R. G., Romascanu, D., Kalbfleisch, C. W., y Waldbusser, S. (2003, agosto). *Introduction to the Remote Monitoring (RMON) Family of MIB Modules* (n.º 3577). RFC 3577. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc3577 doi: 10.17487/RFC3577

Corporation, I. (2020). *In-band network telemetry detects network performance issues* (Inf. Téc.). Intel Network Builders. Descargado de https://builders.intel.com/docs/networkbuilders/in-band-network-telemetry-detects-network-performance-issues.pdf (White Paper)

Corporation, I. (2022). *eBPF offload native mode XDP on intel® ethernet controllers.* https://eci.intel.com/docs/3.0/development/tsnrefsw/bpf-xdp.html.

Corporation, N. (2020). *Accelerating with XDP over mellanox connectx NICs.* https://developer.nvidia.com/blog/accelerating-with-xdp-over-mellanox-connectx-nics/.

DataDog. (2023). *A gentle introduction to XDP.* Descargado de https://www.datadoghq.com/blog/xdp-intro/

DPDK Project. (2018). Pktgen traffic generator using DPDK [Manual de software informático]. Descargado de https://github.com/pktgen/Pktgen-DPDK (Accessed: October 25, 2023)

Elizalde, S., AlSabeh, A., Mazloum, A., Choueiri, S., Kfoury, E., Gomez, J., y Crichigno, J. (2025). A survey on security applications with SmartNICs: Taxonomy, implementations, challenges, and future trends. *Journal of Network and Computer Applications*, *242*, 104257. Descargado de https://www.sciencedirect.com/science/article/pii/S1084804525001547 doi: https://doi.org/10.1016/j.jnca.2025.104257

Embedded.com. (2023). *An introduction to SmartNICs and their role in HPC.* https://www.embedded.com/an-introduction-to-smartnics-and-their-role-in-hpc/.

ESnet. (2019). iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool [Manual de software informático]. Descargado de https://software.es.net/iperf/ (Disponible en línea)

et al., A. (2022). Monarch: Google's planet-scale in-memory time series database. En *Vldb*.

Fedor, M., Schoffstall, M. L., Davin, J. R., y Case, D. J. D. (1990, mayo). *Simple Network Management Protocol (SNMP)* (Inf. Téc. n.º 1157). RFC 1157. Descargado de https://www.rfc-editor.org/info/rfc1157 doi: 10.17487/RFC1157

Google Cloud Blog. (2024, Dec). *Using hubble for GKE dataplane V2 observability.* https://cloud.google.com/blog/products/containers-kubernetes/using-hubble-for-gke-dataplane-v2-observability.

Grafana documentation. (2023). https://grafana.com/docs/.

Gregg, B. (2019). *BPF performance tools: Linux system and application observability.* Boston, MA: Addison-Wesley Professional.

Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., ... Kurien, V. (2015, agosto). Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. , *45*(4), 139–152. doi: https://doi.org/10.1145/2829988.2787496

Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., y Willinger, W. (2018). Sonata: Query-driven streaming network telemetry. En *Proceedings of the 2018 conference of the acm special interest group on data communication* (p. 357–371). New York, NY, USA: Association for Computing Machinery. doi: https://doi.org/10.1145/3230543.3230555

Harrington, D., Wijnen, B., y Presuhn, R. (2002, diciembre). *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks* (n.º 3411). RFC 3411. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc3411 doi: 10.17487/RFC3411

Hastie, T., Tibshirani, R., y Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2.ª ed.). Springer. doi: https://doi.org/10.1007/978-0-387-84858-7

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., y Miller, D. (2018). The express data path: Fast programmable packet processing in the operating system kernel. En *Proceedings of conext '18*. ACM. Descargado de https://doi.org/10.1145/3281411.3281443 doi: 10.1145/3281411.3281443

Infinite Networks. (2023). *SmartNICs: Revolutionizing network performance with infinite networks.* https://infinitenetworksinc.com/smartnics-revolutionizing-network-performance-with-infinite-networks/.

InfluxDB documentation. (2023). https://docs.influxdata.com/.

IO Visor Project. (2023). *eBPF - technology.* https://www.iovisor.org/technology/ebpf.

Jain, R., Rohit, M., Kumar, A., Bakliwal, A., Makwana, A., y Rahevar, M. (2022). Prediction of telemetry data using machine learning techniques. , *11*(9), 1–7. Descargado de https://www.researchgate.net/

publication/363602679_Prediction_of_Telemetry_Data_using_Machine_Learning_Techniques doi: 10.17577/IJERTV11IS090048

Jiang, H., Li, Q., Jiang, Y., Shen, G., Sinnott, R., Tian, C., y Xu, M. (2021). When machine learning meets congestion control: A survey and comparison. *Computer Networks*, *192*. doi: https://doi.org/10.1016/j.comnet.2021.108033

Kandula, S., Sengupta, S., Greenberg, A., Patel, P., y Chaiken, R. (2009). The nature of data center traffic: measurements & analysis. En *Proceedings of the 9th acm sigcomm conference on internet measurement* (p. 202–208). New York, NY, USA: Association for Computing Machinery. doi: https://doi.org/10.1145/1644893.1644918

Kapoor, R., Anastasiu, D. C., y Choi, S. C. (2025). ML-NIC: Accelerating machine learning inference using smart network interface cards. *Frontiers in Computer Science*, *6*, 1493399. doi: https://doi.org/10.3389/fcomp.2024.1493399

Kerrisk, M. (2012, mayo). Namespaces in operation, part 1: namespaces overview. *LWN.net*. Descargado de https://lwn.net/Articles/531114/

Kfoury, E. F., Choueiri, S., Mazloum, A., AlSabeh, A., Gomez, J., y Crichigno, J. (2024). A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access*, *12*, 107297-107336. doi: https://doi.org/10.1109/ACCESS.2024.0429000

Liu, Z., Gao, D., Liu, Y., Zhang, H., y Foh, C. (2017, 07). An adaptive approach for elephant flow detection with the rapidly changing traffic in data center network: An approach for elephant flow detection with the changing traffic. *International Journal of Network Management*, *27*, e1987. doi: https://doi.org/10.1002/nem.1987

McCanne, S., y Jacobson, V. (1993). The BSD packet filter: A new architecture for user-level packet capture. En *Proceedings of the usenix winter 1993 technical conference* (pp. 259–270). San Diego, CA, USA: USENIX Association. Descargado de https://www.tcpdump.org/papers/bpf-usenix93.pdf

Mencagli, G., Torquati, M., Cardaci, A., Fais, A., Rinaldi, L., y Danelutto, M. (2021). Windflow: High-speed continuous stream processing with parallel building blocks. *IEEE Transactions on Parallel and Distributed Systems*, *32*(11), 2748-2763. doi: https://doi.org/10.1109/TPDS.2021.3073970

Miano, S., Lettieri, G., Antichi, G., y Procissi, G. (2024, marzo). Accelerating network analytics with an on-NIC streaming engine. *Comput. Netw.*, *241*(C). doi: https://doi.org/10.1016/j.comnet.2024.110231

Mijumbi, R., Serrat, J., Gorricho, J.-L., Bouten, N., De Turck, F., y Boutaba, R. (2016). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*. Descargado de https://doi.org/10.1109/COMST.2015.2477041 doi: 10.1109/COMST.2015.2477041

namespaces(7) — linux manual page [Manual de software informático]. (2024). Descargado de https://man7.org/linux/man-pages/man7/namespaces.7.html

Netronome Systems, Inc. (2018). Agilio cx smartnic basic firmware user guide [Manual de software informático]. Santa Clara, CA, USA. Descargado de https://help.netronome.com/support/solutions/articles/36000049975-basic-firmware-user-guide (Disponible en línea: Netronome Help Center)

Netronome Systems, Inc. (2018). eBPF offload getting started guide (Revision 1.2 ed.) [Manual de software informático]. Santa Clara, CA. Descargado de https://help.netronome.com/ (Kernel 4.18, Agilio CX SmartNIC)

Netronome Systems, Inc. (2019). Netronome Agilio XC SmartNIC product brief [Manual de software informático]. Cranberry Township, PA, USA. Descargado de https://www.netronome.com/products/agilio-cx/ (Technical documentation and product overview. Accessed: November 2025.)

Nguyen, T. T., y Armitage, G. (2008). A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, *10*(4), 56-76. doi: https://doi.org/10.1109/SURV.2008.080406

Nichols, K., y Jacobson, V. (2018). *Controlled delay active queue management (CoDel)*. RFC 8289, Internet Engineering Task Force (IETF). Descargado de https://datatracker.ietf.org/doc/html/rfc8289

NVIDIA Blog. (2023). *What is a SmartNIC?* https://blogs.nvidia.com/blog/what-is-a-smartnic/.

Panchen, S., McKee, N., y Phaal, P. (2001a, septiembre). *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks* (n.º 3176). RFC 3176. RFC Editor. Descargado de https://www.rfc-editor.org/info/rfc3176 doi: 10.17487/RFC3176

Panchen, S., McKee, N., y Phaal, P. (2001b, septiembre). *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks* (Inf. Téc. n.º 3176). RFC 3176. Descargado de https://www.rfc-editor.org/info/rfc3176 doi: 10.17487/RFC3176

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, É. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, *12*. Descargado de https://www.jmlr.org/papers/v12/pedregosa11a.html

PostgreSQL documentation: Table partitioning. (2023). https://www.postgresql.org/docs/current/ddl-partitioning.html.

The prometheus time series database. (2023). https://prometheus.io/docs/.

Ramakrishnan, K., Floyd, S., y Black, D. (2001). *The addition of explicit congestion notification (ECN) to IP.* RFC 3168, IETF. Descargado de https://datatracker.ietf.org/doc/html/rfc3168

Rizzo, L. (2012). Netmap: a novel framework for fast packet I/O. En *Proceedings of the 2012 usenix conference on annual technical conference* (p. 9). USA: USENIX Association. Descargado de https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf

Roy, A., Zeng, H., Bagga, J., Porter, G., y Snoeren, A. C. (2015, agosto). Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, *45*(4), 123–137. doi: https://doi.org/10.1145/2829988.2787472

Saito, T., y Rehmsmeier, M. (2015). The Precision–Recall plot is more informa-

tive than the ROC plot when evaluating binary classifiers on imbalanced datasets. *PLOS ONE*, *10*(3), e0118432. doi: https://doi.org/10.1371/journal.pone.0118432

Services, A. W. (2023). *Empowering kubernetes observability with eBPF on amazon EKS.* https://aws.amazon.com/blogs/containers/empowering-kubernetes-observability-with-ebpf-on-amazon-eks/.

Sharma, B., y Nadig, D. (2024). ebpf-enhanced complete observability solution for cloud-native microservices. En *Icc 2024 - ieee international conference on communications* (p. 1980-1985). doi: https://doi.org/10.1109/ICC51166.2024.10622329

Silicon Valley Business Journal. (2023). *Intel halts development of tofino switch chips.* Descargado de https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html

Svoboda, J., Ghafir, I., y Prenosil, V. (2015, Oct). Network monitoring approaches: An overview. En *Proc. 3rd international conference on advances in computing, communication and information technology (ccit 2015).* Descargado de https://www.researchgate.net/publication/305957483_Network_Monitoring_Approaches_An_Overview doi: 10.15224/978-1-63248-061-3-72

Systems, C. (2019). *Cisco model-driven telemetry.* https://www.cisco.com/c/en/us/solutions/service-provider/insights/model-driven-telemetry.html.

Tan, L., Su, W., Zhang, W., Lv, J., Zhang, Z., Miao, J., ... Li, N. (2021). In-band network telemetry: A survey. *Computer Networks*, *186*, 107763. doi: https://doi.org/10.1016/j.comnet.2020.107763

TechBlog, N. (2021). *How netflix uses eBPF flow logs at scale for network insight.* https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96.

TimescaleDB as a grafana data source. (2023). https://docs.timescale.com/use-timescale/latest/integrations/grafana/.

TimescaleDB documentation: Hypertables. (2023). https://docs.timescale.com/.

TimescaleDB native compression. (2023). https://docs.timescale.com/use-timescale/latest/compression/.

TimescaleDB whitepaper. (2020). https://www.timescale.com/resources/whitepapers.

Trenton Systems. (2023). *What is a SmartNIC?* https://www.trentonsystems.com/en-us/resource-hub/blog/what-is-a-smartnic.

Wang, S., Balarezo, J. F., Kandeepan, S., Al-Hourani, A., Chavez, K. G., y Rubinstein, B. (2021). Machine learning in network anomaly detection: A survey. *IEEE Access*, *9*, 152379-152396. doi: https://doi.org/10.1109/ACCESS.2021.3126834

Xiong, X., Xie, Y., Chen, X., Zheng, S., Huang, W., y Fengs, J. (2024). An integrated solution for high-efficiency In-band Network Telemetry. En *Proceedings of the acm sigcomm workshop.* New York, NY: ACM. Des-

cargado de https://doi.org/10.1145/3663408.3663425 doi: 10.1145/3663408.3663425

Yaseen, N. (2025). From counters to telemetry: A survey of programmable network-wide monitoring. *Network*, *5*(3). Descargado de https://www.mdpi.com/2673-8732/5/3/38 doi: 10.3390/network5030038

Zhang, C., Patras, P., y Haddadi, H. (2019). Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, *21*(3), 2224-2287. doi: https://doi.org/10.1109/COMST.2019.2904897

Zhou, Y., Sun, C., Liu, H. H., Miao, R., Bai, S., Li, B., ... Xu, M. (2020). Flow event telemetry on programmable data plane. En *Proceedings of the annual conference of the acm special interest group on data communication on the applications, technologies, architectures, and protocols for computer communication* (p. 76–89). New York, NY, USA: Association for Computing Machinery. doi: https://doi.org/10.1145/3387514.3406214

Anexo A

Métricas Recolectadas

A.1. Métricas Recolectadas

Las métricas recolectadas se listan y explican a continuación, así como la tabla a la que pertenecen en la base de datos.

A.1.1. Métricas de la tabla `sys_metrics`

La tabla almacena una captura periódica del estado global del sistema operativo obtenida desde múltiples archivos en /proc. El muestreo continuo crea una serie temporal detallada del comportamiento del host.

- `Name`: Nombre del proceso.
- `State`: Estado actual del proceso (R: ejecución, S: suspendido, D: espera, Z: zombie, T: detenido).
- `Threads`: Número de hilos pertenecientes al proceso.
- `VmRSS`: Memoria residente en RAM (en KB).
- `VmSize`: Memoria virtual total asignada al proceso (en KB).
- `voluntary_ctxt_switches`: Cambios de contexto voluntarios (el proceso cede la CPU).
- `nonvoluntary_ctxt_switches`: Cambios de contexto forzados por el kernel (preempciones).
- `ppid`: PID del proceso padre.
- `utime`: Tiempo de CPU en modo usuario (en ticks).
- `stime`: Tiempo de CPU en modo kernel (en ticks).

- `cutime`: Tiempo de CPU en modo usuario acumulado por procesos hijos (ticks).
- `cstime`: Tiempo de CPU en modo kernel acumulado por procesos hijos (ticks).
- `priority`: Prioridad actual del proceso (relativa al scheduler).
- `nice`: Valor `nice` ajustado por el usuario (impacta en la prioridad).
- `starttime`: Tiempo (en ticks) desde el arranque del sistema hasta la creación del proceso.
- `rss`: Páginas residentes en memoria física.
- `pagesize`: Tamaño de página del sistema (en bytes).
- `read_bytes`: Bytes efectivamente leídos desde disco por el proceso.
- `write_bytes`: Bytes efectivamente escritos a disco por el proceso.
- `nr_switches`: Total de cambios de contexto realizados por el proceso.
- `sum_exec_runtime`: Tiempo total en ejecución del proceso (en segundos).

A.1.2. Métricas de la tabla `proc_metrics`

La tabla proporciona métricas específicas de procesos individuales en el sistema. Su origen es /proc. Esto permite vincular la carga del host con procesos específicos, facilitando atribución de recursos.

- `timestamp`: Marca de tiempo Unix en segundos correspondiente al momento de la captura.
- `MemTotal`: Memoria total disponible en el sistema (kB).
- `MemFree`: Memoria libre actualmente no utilizada (kB).
- `MemAvailable`: Estimación de memoria libre para nuevos procesos, considerando cachés (kB).
- `Buffers`: Memoria utilizada para buffers del kernel.
- `Cached`: Memoria utilizada para cachés de páginas.
- `SwapCached`: Páginas en swap referenciadas recientemente.
- `Active`: Memoria activa en uso por procesos y cachés.
- `Inactive`: Memoria no usada activamente, puede ser liberada.
- `Active(anon)`: Memoria anónima activa.

- `Inactive(anon)`: Memoria anónima inactiva.
- `Active(file)`: Memoria de archivos activa.
- `Inactive(file)`: Memoria de archivos inactiva.
- `Unevictable`: Memoria que no puede ser intercambiada (bloqueada).
- `Mlocked`: Memoria bloqueada por procesos mediante `mlock`.
- `SwapTotal`: Espacio total de swap disponible.
- `SwapFree`: Espacio libre de swap.
- `Dirty`: Páginas modificadas aún no escritas a disco.
- `Writeback`: Páginas en proceso de escritura a disco.
- `AnonPages`: Memoria anónima mapeada.
- `Mapped`: Memoria mapeada a archivos.
- `Shmem`: Memoria compartida (shared memory).
- `Slab`: Memoria utilizada por estructuras internas del kernel.
- `SReclaimable`: Porción recuperable del slab.
- `SUnreclaim`: Parte no recuperable del slab.
- `KernelStack`: Memoria utilizada por pilas de kernel.
- `PageTables`: Memoria utilizada por tablas de páginas.
- `NFS_Unstable`: Páginas en estado de incertidumbre por NFS.
- `Bounce`: Memoria temporal usada en hardware con limitaciones de DMA.
- `WritebackTmp`: Memoria temporal en escritura.
- `CommitLimit`: Límite total de memoria comprometible.
- `Committed_AS`: Memoria comprometida total (incluye sobreasignación).
- `VmallocTotal`: Región total disponible para vmalloc.
- `VmallocUsed`: Espacio de vmalloc en uso.
- `VmallocChunk`: Fragmento contiguo más grande disponible para vmalloc.
- `Percpu`: Memoria asignada por CPU.
- `HardwareCorrupted`: Memoria marcada como corrupta.
- `AnonHugePages`: Memoria anónima en páginas enormes (HugePages).

- `ShmemHugePages`: Memoria compartida asignada en HugePages.
- `ShmemPmdMapped`: Páginas Huge mapeadas vía PMD.
- `CmaTotal`: Total de memoria contigua reservada (CMA).
- `CmaFree`: Memoria contigua libre disponible en CMA.
- `cpu_field_0`: Tiempo de CPU en modo usuario.
- `cpu_field_1`: Tiempo en modo `nice`.
- `cpu_field_2`: Tiempo en modo kernel.
- `cpu_field_3`: Tiempo inactivo.
- `cpu_field_4`: Tiempo de espera por I/O.
- `cpu_field_5`: Tiempo atendiendo interrupciones de hardware.
- `cpu_field_6`: Tiempo atendiendo interrupciones blandas (*softirqs*).
- `cpu_field_7`: Tiempo robado por virtualización.
- `cpu_field_8`: Tiempo ejecutando máquinas virtuales (guest).
- `cpu_field_9`: Tiempo en modo `guest_nice` (si aplica).
- `ctxt`: Total de cambios de contexto desde el arranque.
- `btime`: Tiempo de arranque del sistema (segundos desde Epoch).
- `processes`: Procesos creados desde el arranque.
- `procs_running`: Procesos ejecutándose actualmente.
- `procs_blocked`: Procesos bloqueados esperando I/O.
- `loadavg_1min`: Carga promedio del último minuto.
- `loadavg_5min`: Carga promedio de los últimos cinco minutos.
- `loadavg_15min`: Carga promedio de los últimos quince minutos.
- `loadavg_running`: Número de procesos actualmente activos.
- `loadavg_total`: Total de procesos en el sistema.
- `loadavg_last_pid`: Último PID asignado.
- `uptime_total`: Segundos desde el arranque del sistema.
- `uptime_idle`: Tiempo inactivo acumulado de todos los CPUs.
- `<iface>_rx_bytes`: Bytes recibidos por la interfaz `<iface>`.

- `<iface>_rx_packets`: Paquetes recibidos.
- `<iface>_tx_bytes`: Bytes transmitidos.
- `<iface>_tx_packets`: Paquetes transmitidos.
- `<disk>_read_ios`: Operaciones de lectura completadas.
- `<disk>_read_sectors`: Sectores leídos.
- `<disk>_write_ios`: Operaciones de escritura completadas.
- `<disk>_write_sectors`: Sectores escritos.
- `tcp_active`: Conexiones TCP activas iniciadas localmente.
- `tcp_passive`: Conexiones TCP pasivas aceptadas.
- `tcp_failed`: Intentos fallidos de conexión.
- `tcp_established`: Conexiones establecidas activas.
- `udp_in`: Paquetes UDP recibidos.
- `udp_out`: Paquetes UDP enviados.
- `icmp_in`: Mensajes ICMP recibidos.
- `icmp_out`: Mensajes ICMP enviados.
- `ip_in`: Paquetes IP recibidos.
- `ip_out`: Paquetes IP enviados.

A.1.3. Métricas de la tabla `net_metrics`

Esta tabla almacena estadísticas globales de red capturadas periódicamente desde el datapath eBPF. Los valores representan deltas del último intervalo de captura (100 ms). Sirve para caracterizar la carga y composición del tráfico a nivel del host/red, sin considerar identidades de flujo.

- `ts` (`timestamptz`): Marca temporal de la muestra en UTC.
- `pps` (`bigint`): Paquetes observados en el último intervalo ($\approx$ 100 ms).
- `size_class_0` (`bigint`): Paquetes en el último intervalo en la clase de tamaño 0.
- `size_class_1` (`bigint`): Paquetes en el último intervalo en la clase de tamaño 1.
- `size_class_2` (`bigint`): Paquetes en el último intervalo en la clase de tamaño 2.

- `size_class_3` (`bigint`): Paquetes en el último intervalo en la clase de tamaño 3.
- `size_class_4` (`bigint`): Paquetes en el último intervalo en la clase de tamaño 4.
- `fin` (`bigint`): Conteo de paquetes con flag TCP FIN en el último intervalo.
- `syn` (`bigint`): Conteo de paquetes con flag TCP SYN en el último intervalo.
- `rst` (`bigint`): Conteo de paquetes con flag TCP RST en el último intervalo.
- `psh` (`bigint`): Conteo de paquetes con flag TCP PSH en el último intervalo.
- `ack` (`bigint`): Conteo de paquetes con flag TCP ACK en el último intervalo.
- `urg` (`bigint`): Conteo de paquetes con flag TCP URG en el último intervalo.
- `eth_ip` (`bigint`): Tramas Ethernet con tipo IP (IPv4/IPv6 según clasificación) observadas en el último intervalo.
- `eth_arp` (`bigint`): Tramas Ethernet con tipo ARP observadas en el último intervalo.
- `ip_tcp` (`bigint`): Paquetes IP cuyo protocolo es TCP, contados en el último intervalo.
- `ip_udp` (`bigint`): Paquetes IP cuyo protocolo es UDP, contados en el último intervalo.
- `ecn_ce` (`bigint`): Paquetes con marca ECN-CE observados en el último intervalo.

A.1.4. Métricas de la tabla `flow_metrics_logs`

Registro histórico de actividad por flujo TCP (identificado por una 6-tupla) en instantes discretos. Se utiliza para entrenar modelos de ML, detección de congestión y análisis de dinámica temporal del tráfico.

Aclaraciones:

- Las claves de flujo siguen la estructura `flow_key`: {`src_ip`, `dst_ip`, `src_port`, `dst_port`, `protocol`}.
- `flow_let` identifica el *subflujo* o *flowlet* dentro del mismo flujo de 5-tuplas.
- `last_timestamp_ns` y `delta_ns` provienen del mapa `flow_timer`; el delta se calcula con reloj monótonico en nanosegundos.

- `packets` y `bytes` provienen de mapas eBPF de conteo (`flow_cant`/`flow_bytes`).
- Flags (`FIN`, `SYN`, `RST`, `PSH`, `ACK`, `URG`) y `CE` provienen de `flow_flags_map`.
- Si no hay datos en `flow_stats`, los derivados (*throughput*, *loss_est*, *dup_acks_est*) se registran en cero.

Campos de `flow_metrics_logs`

- `ts` (`timestamptz`): Marca temporal en UTC de la captura.
- `pid` (`int`): PID asociado al flujo si fue detectado; -1 si no se resolvió.
- `src_ip` (`inet`): Dirección IP de origen.
- `dst_ip` (`inet`): Dirección IP de destino.
- `src_port` (`int`): Puerto de origen.
- `dst_port` (`int`): Puerto de destino.
- `protocol` (`int`): Protocolo de capa 4 (p. ej., 6=TCP, 17=UDP).
- `flow_let` (`bigint`): Identificador de subflujo (flowlet) actual.
- `last_timestamp_ns` (`bigint`): Timestamp monótonico (ns) del último paquete visto para la 5-tupla.
- `delta_ns` (`bigint`): Diferencia monótonica (ns) entre el momento actual y `last_timestamp_ns`.
- `packets` (`bigint`): Cantidad de paquetes observados para la clave en la última lectura.
- `delta_packets` (`bigint`): Diferencia entre paquetes observados para la clave entre la actual y la última lectura.
- `bytes` (`bigint`): Cantidad de bytes observados para la clave en la última lectura.
- `fin` (`bigint`): Conteo de paquetes con flag TCP FIN para la clave.
- `syn` (`bigint`): Conteo de paquetes con flag TCP SYN para la clave.
- `rst` (`bigint`): Conteo de paquetes con flag TCP RST para la clave.
- `psh` (`bigint`): Conteo de paquetes con flag TCP PSH para la clave.
- `ack` (`bigint`): Conteo de paquetes con flag TCP ACK para la clave.
- `urg` (`bigint`): Conteo de paquetes con flag TCP URG para la clave.
- `throughput` (`double precision`): *Throughput* estimado en Mbps. Se toma de `flow_stats` como `throughput_mbps_x100`/100.

- `loss_est_pkts` (`bigint`): Estimación de pérdida de paquetes por flujo; se calcula como

  $$\text{máx}\big((\texttt{loss\_dup\_pkts\_a}+\texttt{loss\_dup\_pkts\_b})-(\texttt{loss\_ofo\_pkts\_a}+\texttt{loss\_ofo\_pkts\_b}),\ 0\big).$$

- `dup_acks_est` (`bigint`): Estimación de *duplicate ACKs* por flujo; se aproxima con

  $$\texttt{loss\_ofo\_pkts\_a} + \texttt{loss\_ofo\_pkts\_b}.$$

- `ce` (`bigint`): Conteo de paquetes marcados con ECN-CE asociados al flujo.

A.1.5. Métricas de la tabla `tcp_subflows_live`

La tabla `tcp_subflows_live` mantiene una **vista viva** del estado de cada subflujo TCP (*flowlet*), actualizada en tiempo real por el programa `visor.c` mediante un `UPSERT`. Cada fila representa la última información disponible para una 5-tupla más el identificador de subflujo.

Conceptos

- **Clave primaria**: (`src_ip`, `dst_ip`, `src_port`, `dst_port`, `protocol`, `subflow_id`).
- `subflow_id` equivale a `flow_let` y permite dividir un flujo largo en segmentos temporalmente coherentes.
- La columna `last_seen` indica el instante en que se recibió el último paquete del subflujo.

Campos de `tcp_subflows_live`

- `src_ip` (`inet`): Dirección IP origen del flujo.
- `dst_ip` (`inet`): Dirección IP destino del flujo.
- `src_port` (`int`): Puerto de origen.
- `dst_port` (`int`): Puerto de destino.
- `protocol` (`int`): Protocolo de capa 4.
- `subflow_id` (`bigint`): Identificador del subflujo asociado a la 5-tupla.
- `pid` (`int`): PID del proceso local si se conoce; -1 en caso contrario.
- `last_seen` (`timestamptz`): Último instante en el que el subflujo estuvo activo.
- `packets` (`bigint`): Paquetes totales observados en este subflujo.
- `bytes` (`bigint`): Bytes totales observados en este subflujo.

- `fin` (`bigint`): Cantidad acumulada de paquetes FIN.
- `syn` (`bigint`): Cantidad acumulada de paquetes SYN.
- `rst` (`bigint`): Cantidad acumulada de paquetes RST.
- `psh` (`bigint`): Cantidad acumulada de paquetes PSH.
- `ack` (`bigint`): Cantidad acumulada de paquetes ACK.
- `urg` (`bigint`): Cantidad acumulada de paquetes URG.
- `snd_cwnd` (`int`): Ventana local de envio (en segmentos).
- `rtt_ms` (`double precision`): RTT estimado en milisegundos.
- `bytes_acked` (`bigint`): Bytes confirmados según ACK recibidos.
- `bytes_retrans` (`bigint`): Bytes retransmitidos.
- `retrans_pkts` (`int`): Paquetes retransmitidos.
- `throughput_mbps` (`double precision`): Throughput estimado en Mbps.
- `rwnd_local_kb` (`double precision`): Receive Window local (kB).
- `rwnd_peer_kb` (`double precision`): Receive Window del peer (kB).
- `snd_wscale` (`smallint`): Escala de ventana enviada.
- `rcv_wscale` (`smallint`): Escala de ventana recibida.
- `rx_dup_pkts_a`, `rx_dup_pkts_b`: Paquetes duplicados detectados en ambos sentidos.
- `rx_dup_bytes_a`, `rx_dup_bytes_b`: Bytes duplicados.
- `rx_ofo_pkts_a`, `rx_ofo_pkts_b`: Paquetes fuera de orden (Out-of-order).
- `rx_ofo_bytes_a`, `rx_ofo_bytes_b`: Bytes fuera de orden.
- `ce` (`bigint`): Paquetes con marca ECN-CE para el subflujo.

Tabla A.1: Principales métricas recolectadas por el sistema de monitoreo.

Campo	**Tabla**	**Nivel**	**Descripción breve**
Name	sys_metrics	Host / procesos	Nombre del proceso monitoreado.
VmRSS	sys_metrics	Host / procesos	Memoria residente en RAM utilizada por el proceso (kB).
sum_exec_runtime	sys_metrics	Host / procesos	Tiempo total de ejecución del proceso (s).
timestamp	proc_metrics	Host / global	Marca de tiempo de la captura del estado del sistema.
MemAvailable	proc_metrics	Host / memoria	Memoria disponible para procesos (kB).
cpu_field_0–3	proc_metrics	Host / CPU	Tiempos acumulados en usuario, nice, kernel e inactivo.
loadavg_1min	proc_metrics	Host / carga	Carga promedio del último minuto.
ts	net_metrics	Red / global	Marca temporal de la muestra de tráfico.
pps	net_metrics	Red / global	Paquetes observados en el último intervalo.
size_class_0–4	net_metrics	Red / global	Distribución de paquetes por clase de tamaño.
ecn_ce	net_metrics	Red / global	Paquetes marcados con ECN-CE en el intervalo.
ts	flow_metrics_logs	Flujo TCP (histórico)	Marca temporal de la lectura del flujo.
src_ip, dst_ip	flow_metrics_logs	Flujo TCP (histórico)	Direcciones IP de la 5-tupla del flujo.
flow_let	flow_metrics_logs	Flujo TCP (histórico)	Identificador del subflujo.
packets, bytes	flow_metrics_logs	Flujo TCP (histórico)	Paquetes y bytes acumulados observados.
delta_ns	flow_metrics_logs	Flujo TCP (histórico)	Intervalo temporal entre lecturas consecutivas (ns).
ecn_ce	flow_metrics_logs	Flujo TCP (histórico)	Conteo ECN-CE asociado al flujo.
throughput_mbps	flow_metrics_logs	Flujo TCP (histórico)	Throughput estimado del flujo en Mbps.
subflow_id	tcp_subflows_live	Subflujo TCP (vivo)	Identificador del subflujo activo.
packets, bytes	tcp_subflows_live	Subflujo TCP (vivo)	Paquetes y bytes acumulados.
rtt_ms	tcp_subflows_live	Subflujo TCP (vivo)	RTT estimado (ms).
ce	tcp_subflows_live	Subflujo TCP (vivo)	Paquetes marcados con ECN-CE en el subflujo.

Anexo B

Cálculos sobre Métricas

Se muestran las formulas utilizadas para el porcentaje de uso de CPU, el porcentaje de memoria y la carga promedio.

Cada fila de la tabla (`sys_metrics`) corresponde a un conjunto de valores observados en un tiempo t_i con un campo temporal `timestamp`.

Se definen las siguientes variables base:

- $cpu\_field_0 = \text{user}_t$: Tiempo que el CPU estuvo ejecutando procesos en modo usuario
- $cpu\_field_1 = \text{nice}_t$: Tiempo en modo usuario con prioridad reducida. Corresponde a tareas en segundo plano o de baja prioridad.
- $cpu\_field_2 = \text{system}_t$: Tiempo ejecutando código en modo kernel (llamadas al sistema, controladores, interrupciones de software).
- $cpu\_field_3 = \text{idle}_t$: Tiempo en que el CPU estuvo ocioso (sin ejecutar procesos) y no esperando I/O.
- $cpu\_field_4 = \text{iowait}_t$: Tiempo que el CPU permaneció esperando operaciones de entrada/salida (I/O) — por ejemplo, lecturas de disco o red. El CPU está inactivo pero el proceso espera datos.
- $cpu\_field_5 = \text{irq}_t$: Tiempo consumido en interrupciones de hardware, es decir, atendiendo señales de dispositivos (NIC, discos, timers, etc.).
- $cpu\_field_6 = \text{softirq}_t$: Tiempo ejecutando interrupciones de software (softirqs).
- $memtotal_t$ = memoria total del sistema (kB)
- $memavailable_t$ = memoria disponible estimada (kB)
- $loadavg\_1min_t$ = carga promedio en el último minuto

Todas las variables de CPU son contadores acumulativos desde el arranque del sistema, expresados en *ticks* de reloj o *jiffies*. Por tanto, los valores instantáneos deben calcularse a partir de diferencias entre muestras consecutivas.

Uso de CPU (%)

Hipótesis.

1. Los campos de `/proc/stat` representan el tiempo total acumulado de CPU en distintas categorías desde el arranque.
2. La proporción de tiempo activo del CPU durante un intervalo $(t_{i-1}, t_i]$ se puede obtener restando el tiempo ocioso (`idle`) del tiempo total transcurrido.
3. La muestra se realiza con intervalo constante Δt suficientemente pequeño como para considerar que el uso es constante dentro del intervalo.

Desarrollo. Definimos el tiempo total acumulado hasta el instante t_i como:

$$T_i = user_i + nice_i + system_i + idle_i + iowait_i + irq_i + softirq_i$$

y el tiempo ocioso acumulado como:

$$I_i = idle_i$$

El incremento de tiempo total y de tiempo ocioso entre dos muestras consecutivas es:

$$\Delta T_i = T_i - T_{i-1}$$

$$\Delta I_i = I_i - I_{i-1}$$

La fracción de tiempo *ocupado* (no ocioso) en el intervalo $(t_{i-1}, t_i]$ es:

$$f_i = \frac{\Delta T_i - \Delta I_i}{\Delta T_i}$$

Por lo tanto, el **porcentaje de uso del CPU** en ese intervalo se define como:

$$\boxed{CPU\_usage_i = 100 \times \frac{\Delta T_i - \Delta I_i}{\Delta T_i}}$$

Esta expresión equivale a:

$$CPU\_usage_i = 100 \times \left(1 - \frac{\Delta I_i}{\Delta T_i}\right)$$

Uso de RAM (%)

Hipótesis.

1. Los campos de `/proc/meminfo` `MemTotal` y `MemAvailable` se expresan en kilobytes.

2. `MemAvailable` representa una estimación realista de la memoria que puede asignarse a nuevos procesos sin necesidad de swap, por lo que la diferencia $(MemTotal - MemAvailable)$ aproxima la memoria realmente usada.

Desarrollo. Sea la memoria total del sistema M_T y la memoria disponible M_A. La memoria utilizada es:

$$M_U = M_T - M_A$$

Por tanto, el porcentaje de uso de memoria se define como:

$$\boxed{RAM\_usage_i = 100 \times \frac{M_T - M_A}{M_T}}$$

Esta expresión considera tanto la memoria usada por procesos como la utilizada en cachés y buffers que no pueden liberarse inmediatamente.

Carga promedio del sistema (Load Average)

Hipótesis.

1. El archivo `/proc/loadavg` mantiene un promedio exponencial móvil del número de procesos en ejecución o listos para ejecutarse (estado R o D).

2. Los tres valores reportados corresponden a ventanas de 1, 5 y 15 minutos.

Desarrollo. Denotando por $L_{1,i}$ el valor leído en el campo `loadavg_1min` en el instante t_i, se tiene:

$$\boxed{LoadAvg\_1min_i = L_{1,i}}$$

No requiere procesamiento adicional, pues el kernel ya actualiza continuamente la media móvil según la expresión:

$$L_{1,t} = L_{1,t-\Delta t} \times e^{-\frac{\Delta t}{\tau}} + n_t \times \left(1 - e^{-\frac{\Delta t}{\tau}}\right)$$

donde τ es la constante de tiempo (1 minuto) y n_t el número instantáneo de procesos activos o bloqueados.

Expresiones finales

$$
\begin{aligned}
CPU\_usage_i &= 100 \times \frac{(T_i - T_{i-1}) - (I_i - I_{i-1})}{T_i - T_{i-1}} \\
RAM\_usage_i &= 100 \times \frac{M_T - M_A}{M_T} \\
LoadAvg\_1min_i &= L_{1,i}
\end{aligned}
$$

Interpretación.

- CPU_usage_i mide la proporción de tiempo en que el CPU estuvo ejecutando procesos de usuario o kernel, excluyendo los periodos de inactividad.
- RAM_usage_i refleja la fracción de memoria ocupada por procesos y datos residentes en RAM.
- $LoadAvg\_1min_i$ expresa la carga de trabajo promedio del sistema, es decir, la cantidad media de tareas listas o en ejecución durante el último minuto.