Clasificación de flujos elefante y ratón en planos de datos programables

Trabajo de Grado

David Camilo Muñoz Garcia
Freddy Andres Saavedra Hoyos

Director: PhD. Oscar Mauricio Caicedo Rendón
Codirector: PhD. Felipe Estrada Solano

Departamento de Telemática
Facultad de Ingeniería Electrónica y Telecomunicaciones
Universidad del Cauca
Popayán, Cauca, 2022

Clasificación de flujos elefante y ratón en planos de datos programables

David Camilo Muñoz Garcia
Freddy Andres Saavedra Hoyos

Trabajo de grado presentado a la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca para obtener el título de:
Ingeniero en Electrónica y Telecomunicaciones

Director: PhD. Oscar Mauricio Caicedo Rendón
Codirector: PhD. Felipe Estrada Solano

Departamento de Telemática
Facultad de Ingeniería Electrónica y Telecomunicaciones
Universidad del Cauca
Popayán, Cauca, 2022

Agradecimientos

Primero, agradecer a Dios, a nuestras familias, especialmente a nuestros padres, madres y hermanos por su apoyo a lo largo del desarrollo de nuestra vida personal y académica. Agradecer a todas las personas que hicieron parte de nuestra vida universitaria, incluyendo amigos, compañeros del programa de Ingeniería Electrónica y Telecomunicaciones, los cuales fueron un pilar y apoyo importante en toda nuestra etapa formativa, y les deseamos salud, prosperidad y buena fortuna. Agradecer a nuestro director Oscar Mauricio Caicedo, quien con toda su paciencia y dedicación nos guio durante todo el proceso de nuestro trabajo de grado, exaltamos sus excelentes cualidades académicas, técnicas y humanas. Mencionar, además, al semillero de investigación COMSOCAUCA y en especialmente a nuestro codirector Felipe Estrada Solano quienes nos brindaron las bases teóricas para la realización de este trabajo de grado. Finalmente, nos sentimos orgullosos por haber realizado nuestra formación académica en la prestigiosa Universidad del Cauca, donde logramos crear lazos de amistad que perdurarán por toda nuestra vida, conocer profesionales con formaciones y valores humanos excelentes, convirtiéndose así en el lugar donde logramos cumplir metas y sueños

Resumen

El gran volumen, variedad y velocidad del tráfico generado por el advenimiento de IoT (*Internet of Things*) y la inclusión exponencial de nuevas aplicaciones y servicios en la red, requieren soporte en los centros de datos. Las técnicas de enrutamiento convencionales en los centros de datos como ECMP (*Equal Cost Multi-Path*) pueden degradar el rendimiento de la red cuando manejan flujos elefante y ratón. Lo cual compromete el cumplimiento de los requisitos de latencia, QoS (*Quality of Service*) y QoE (*Quality of Experience*) del centro de datos, y afecta directamente el desempeño de las aplicaciones y servicios que soporta. Por esta razón, se han propuesto novedosas técnicas para clasificar los flujos elefante y procesarlos especialmente en pro de la red. Varios enfoques emplean ML (*Machine Learning*) en el Plano de Control SDN (*Software Defined Networking*) o en los servidores de la red para clasificar los flujos. Sin embargo, estos métodos producen sobrecarga de tráfico, baja escalabilidad y alto tiempo de clasificación. Este Trabajo de grado propone un nuevo método de clasificación de flujos elefante y ratón, llamado P4Tree, el cual aplica RF (*Random Forest*) en el Plano de Datos programable de una red SDN para clasificar los flujos de forma rápida y precisa. P4Tree entrena un modelo RF fuera de línea y lo despliega en el Plano de Datos para que clasifique flujos a velocidad de línea. P4Tree es exhaustivamente evaluado a partir de trazas de tráfico real y diferentes configuraciones de RF. Los resultados de la evaluación muestran que P4Tree es preciso y logra un tiempo de clasificación bajo.

Índice General

Índice de Figuras VII

Índice de Tablas VIII

Lista de Abreviaciones IX

1. Introducción 1

1.1. Objetivos 4

1.1.1. Objetivo General 4

1.1.2. Objetivos Específicos 4

1.2. Contribuciones 4

1.3. Organización 5

2. Conceptos Fundamentales 6

2.1. Red de Centro de Datos 6

2.2. Clasificación de Flujos 8

2.3. Plano de Datos Programable 10

2.3.1. Procesadores de Paquetes Independientes del Protocolo de Programación 10

2.3.2. Programación de Procesadores de Paquetes Independientes del Protocolo 13

2.3.3. Machine Learning en Planos de Datos Programables 15

3. Trabajos Relacionados 20

3.1. Clasificación de Flujos en el Plano de Control 20

3.2. Clasificación de Flujos en los Servidores 22

3.3. Clasificación de Flujos en el Plano de Datos 23

3.4. Clasificación de Flujos con Esquemas Híbridos 25

3.5. Brechas 27

4. Mecanismo basado en Random Forest para Clasificar Flujos Elefante y Ratón en Planos de Datos Programables 29

4.1. Arquitectura de P4Tree 30

4.2. Entrenamiento de P4Tree 32

4.2.1. Extracción de Características 32

4.2.2. Construcción del Modelo RF 33

4.2.3. Mapeo de Modelo RF a entradas *match-action* 33

4.3. Inferencia de P4Tree 35

4.3.1. Filtro 36

4.3.2. Monitor 37

4.3.3. Clasificador 38

4.3.4. Notificador 38

5. Evaluación y Análisis de Resultados 41

5.1. Prototipo 41

5.2. Datasets 43

5.3. Métricas de Rendimiento 44

5.3.1. Métricas de Precisión 44

5.3.2. Tiempo de Etiquetado 46

5.3.3. Paquetes Eliminados 46

5.4. Configuración del Experimento 47

5.5. Análisis de Paquetes Eliminados 48

5.6. Análisis de Métricas de Precisión 49

5.7. Análisis de Tiempo de Etiquetado 51

5.8. Observaciones 52

5.9. Análisis Comparativo 53

5.9.1. Técnica de Clasificación 53

5.9.2. Elefantes Detectados 54

5.9.3. Elefantes Falsos 55

5.9.4. Tiempo de Clasificación 55

5.9.5. Modificaciones de Red 56

5.9.6. Factores de Desempeño 56

6. Conclusiones y Trabajos Futuros **57**

6.1. Conclusiones 57

6.2. Trabajos Futuros 58

Bibliografía **58**

Anexos **71**

A. Prototipo **1**

B. Publicación **2**

Índice de Figuras

2.1. Topología de Red: *Fat-tree* 7

2.2. Arquitectura de *switch* PISA 10

2.3. Simple topología de un *switch* PISA con 4 servidores conectados . . . 11

2.4. Proceso de ejecución de un programa P4 en un *switch* PISA, adaptada de [1] 14

2.5. Proceso para desplegar ML en el Plano de Datos 17

2.6. Random Forest 18

4.1. Arquitectura de P4Tree 31

5.1. Prototipo de P4Tree 42

5.2. Matriz de Confusión 45

5.3. Test de Estrés. Se resalta la velocidad original de UNIV1 y UNIV2. . 48

5.4. Precisión de P4Tree con un modelo RF de 1 árbol (izquierda) y un modelo RF de 9 árboles (derecha) al variar los pesos de los flujos elefante W_1 para UNIV1 (arriba) y UNIV2 (abajo) 50

Índice de Tablas

2.1. Métricas para clasificar flujos según Lan y Heidemann [2] 9

3.1. Brechas 28

4.1. Símbolos en la arquitectura P4Tree 32

5.1. Detalles de las trazas de tráfico real y los flujos IPV4 obtenidos utilizando el encabezado 5-tuplas y $\theta_{TO} = 5$s 43

5.2. Configuración de entrenamiento para RF en Scikit-Learn 47

5.3. Métricas de Precisión 49

5.4. Tiempo de Etiquetado 51

5.5. Comparación de P4Tree con trabajos relacionados 53

Lista de Abreviaciones

AFL *Asynchronous Federated Learning* - Aprendizaje Federado Asíncrono

AHT *Adaptive Hoeffding Tree* - Árbol Hoeffding Adaptativo

AHOT *Adaptive Hoeffding Option Tree* - Árbol Hoeffding de Opciones Adaptativo

ANN *Artificial Neural Network* - Red Neuronal Artificial

ARF *Adaptive Random Forest* - Bosque Aleatorio Adaptativo

BNN *Binary Neural Network* - Red Neuronal Binaria

BMV2 *Behavior Model Version 2* - Modelo de Comportamiento Versión 2

CMS *Count Min Sketch* - Croquis de Conteo Mínimo

DCN *Data Center Networks* - Redes de Centro de Datos

DDoS *Distributed Denial of Service* - Denegación Distribuida de Servicio

DoS *Denial-of-Service* - Denegación de Servicio

DNN *Deep Neural Network* - Red Neuronal Profunda

DRL *Deep Reinforcement Learning* - Aprendizaje por Refuerzo Profundo

DTL *Dynamical Traffic Learning* - Aprendizaje de Tráfico Dinámico

ECMP *Equal Cost Multi-Path* - Múltiples Rutas de Igual Costo

FCT *Flow Completion Time* - Tiempo de Finalización de Flujo

FN *False Negatives* - Falsos Negativos

FP *False Positives* - Falsos Positivos

FPR *False Positive Rate* - Tasa de Falsos Positivos

GRU *Gated Recurrent Unit* - Unidad Recurrente Cerrada

gRPC *google Remote Procedure Call* - Llamada de Procedimiento Remoto de google

IAT *Inter-Arrival Time* - Tiempo Entre Llegadas

IoT *Internet of Things* - Internet de las Cosas

KM *K-Means* - K-Medias

MCC *Matthews Correlation Coefficient* - Coeficiente de Correlación de Matthews

NB *Naïve Bayes* - Bayesiano Ingenuo

OPP *Open Packet Processor* - Procesador de Paquetes Abiertos

OSPF *Open Shortest Path First* - Abrir el Camino más Corto Primero

PSD *Package Size Distribution* - Distribución del Tamaño del Paquete

PISA *Protocol Independent Switch Architecture* - Arquitectura de Switch Independiente del Protocolo

PTT *Packet Processing Time* - Tiempo de Procesamiento de Paquetes

Protobuf *Protocol Buffers* - Protocolo de Búferes

QoE *Quality of Experience* - Calidad de Experiencia

QoS *Quality of Service* - Calidad de Servicio

RF *Random Forest* - Bosque Aleatorio

RL *Reinforcement Learning* - Aprendizaje por Refuerzo

SDN *Software Defined Networking* - Redes Definidas por Software

SVM *Support Vector Machine* - Máquinas de Vector de Soporte

TE *Traffic Engineering* - Ingeniería de Tráfico

TM *Template Matching* - Comparación de Plantillas

TN *True Negatives* - Verdaderos Negativos

ToR *Top-of-Rack* - Parte Superior del Rack

TP *True Positives* - Verdaderos Positivos

TPR *True Positive Rate* - Tasa de Verdaderos Positivos

VFDT *Very Fast Decision Tree* - Árbol de Decisión Muy Rápido

WCMP *Weighted Cost Multi-Path* - Múltiples Rutas de Costo Ponderado

Capítulo 1

Introducción

Internet se ha convertido en una necesidad básica para la población que tiene acceso a la tecnología, genera un consumo masivo y constante de los productos y servicios que ahí se alojan, esto la constituye como la más grande y eficiente "máquina" de hacer dinero en la actualidad [3]. El consumismo online crea la necesidad de innovar, por ende todos los días se indexan aplicaciones y servicios nuevos en la red [4][5], incrementando altamente el tráfico que circula en la misma y con ello el uso de recursos informáticos y de almacenamiento. El tráfico será aún más relevante en términos de variedad, velocidad y volumen, con el advenimiento del IoT, que para el año 2025, pretende conectar 75 mil millones de dispositivos a la red [6][7] (desde bombillos y electrodomésticos para hogares inteligentes, hasta máquinas industriales y carros autónomos). IoT es soportado por el ancho de banda proporcionado por las DCN (*Data Center Networks*) [8], que actualmente, proyectan su infraestructura para suplir la monumental e inminente demanda de conectividad [9].

Evitar la congestión del creciente tráfico en la red, es uno de los principales desafíos de TE[1] (*Traffic Engineering*) que comparten las redes convencionales y las DCN [11][12]. En ellas, protocolos tradicionales como ECMP [13], WCMP (*Weighted Cost Multi-Path*) [14] y OSPF (*Open Shortest Path First*) [15], enrutan sin tener en

[1]La ingeniería de tráfico es un conjunto de técnicas que buscan mejorar el rendimiento de la red, la QoS y la experiencia del usuario, mediante el uso de métodos para medir y administrar el tráfico eficientemente, lo que también puede reducir los costos operativos [10].

cuenta la utilización de los enlaces. Además, no discriminan entre flujos elefante (*i.e.*, grandes y longevos) y flujos ratón (*i.e.*, pequeños y cortos) [16][17], generando congestión en algunos enlaces por destinar más elefantes sobre la misma ruta (*i.e.*, puntos calientes). Dicha situación compromete el desempeño, la latencia, la QoS y QoE de aplicaciones (*e.g.*, control industrial, control remoto, etc.), servicios (*e.g.*, *streaming*, videojuegos en línea, etc.) y en un futuro, de las aplicaciones enmarcadas en los casos de uso de 5G (*e.g.*, cirugía remota, conducción autónoma, etc.) [18][19]. En este contexto, la clasificación de flujos elefante entra a jugar un papel importante para TE [20][21]. En especial, para mecanismos de enrutamiento inteligente de flujos elefante [22][23], mecanismos de enrutamiento inteligente de tráfico sensible a la latencia [24][25] y balanceadores de carga [26][27].

La versatilidad y programabilidad de SDN permite clasificar flujos elefante en diferentes planos de la red [28]. Los trabajos [29][30][31] clasifican los flujos en el Plano de Control usando técnicas basadas en ML. Estas contribuciones obtienen una buena precisión en la clasificación de los elefantes, sin embargo, es necesario recolectar información de los flujos en los *switches*, lo cual introduce un alto tiempo de clasificación y sobrecarga (*i.e.*, sobrecarga de comunicación entre los *switches* y el controlador). Un tiempo de clasificación alto retrasa la toma de decisiones en los mecanismos que dependen de la clasificación de flujos elefante [24][22][26][27], degradando su rendimiento. Trabajos como [32][33] proponen migrar la clasificación de flujos a los servidores de la red. Estos métodos reducen el tiempo de clasificación y la sobrecarga sin sacrificar la precisión. Sin embargo, operan en el kernel del sistema operativo de los servidores con el objetivo de monitorear todos los paquetes enviados por las aplicaciones, contenedores y máquinas virtuales [32]. Lo cual genera problemas de escalabilidad, ya que es necesario modificar el sistema operativo de cada servidor de la red para llevar a cabo una implementación. Además, su correcto funcionamiento depende directamente de los recursos de los servidores (*i.e.*, procesamiento y almacenamiento).

En el Plano de Datos, trabajos como [34][35][36][37][38] aprovechan PISA (*Protocol Independent Switch Architecture*) [39] para clasificar flujos en los *switches* programables de la red. Esta nueva arquitectura de switch admite lenguajes como P4 [40] y Lucid [41] para programar los *switches*. Lo que beneficia la tarea de clasificación,

ya que se puede manejar desde el borde de la red a velocidad de línea[2] y con algoritmos ML compatibles con *switches* programables [42], es decir, que no usan números flotantes ni bucles para clasificar. En particular, los trabajos [36][37][38] proponen clasificadores basados en contadores de paquetes. Estos clasificadores logran una precisión casi perfecta a costa de un alto consumo de memoria. Lo cual se debe a los largos períodos de tiempo que la información del flujo debe almacenarse dentro del *switch* (*i.e.*, hasta que el flujo exceda un umbral de tamaño o número de paquetes establecido). Además, al igual que el único trabajo (hasta donde sabemos) que clasifica flujos elefante con ML en el *switch* [35], el tiempo de clasificación es alto. Otros enfoques híbridos funcionan en un Plano de Datos programable y se apoyan en el controlador para corroborar las decisiones tomadas con flujos específicos [43][44][45]. Esto aumenta la precisión y la flexibilidad del mecanismo para cualquier tipo de tráfico, sin embargo, la inclusión del controlador implica tiempo de clasificación y sobrecarga adicionales.

A pesar del progreso en la clasificación de flujos elefante y ratón en el Plano de Datos, hasta donde sabemos, ningún trabajo considera el balance entre precisión y tiempo de clasificación. Por tanto, este trabajo de grado busca resolver la siguiente pregunta de investigación:

¿Cómo clasificar flujos elefante y ratón en Planos de Datos programables considerando el balance entre precisión y tiempo de clasificación?

Para abordar la pregunta de investigación, este trabajo plantea la siguiente hipótesis: Utilizar ML en Planos de Datos programables permite una clasificación de flujos elefante y ratón, equilibrando la precisión y el tiempo de clasificación.

[2]Clasificar a velocidad de línea ó en línea, hace referencia a realizar dicha acción mientras se procesan paquetes en tiempo real.

1.1. Objetivos

1.1.1. Objetivo General

- Proponer un mecanismo para clasificar flujos elefante y ratón en Planos de Datos programables considerando el balance entre precisión y tiempo de clasificación.

1.1.2. Objetivos Específicos

- Diseñar un mecanismo para clasificar flujos elefante y ratón en Planos de Datos programables.
- Implementar el mecanismo de referencia en un entorno emulado.
- Evaluar la eficiencia de la implementación en términos de precisión y tiempo de clasificación de flujos.

1.2. Contribuciones

Este trabajo de grado tiene como objetivo lograr los siguientes aportes:

- Un mecanismo para clasificar flujos elefante y ratón en Planos de Datos programables que considere el balance entre precisión y tiempo de clasificación.
- La implementación y evaluación del mecanismo propuesto.
- Conocimiento relacionado con programabilidad en el Plano de Datos para la línea de investigación en servicios avanzados de telecomunicaciones.

El trabajo presentado en esta monografía fue preparado para enviar a la comunidad científica a través de un artículo a una revista indexada (ver Anexo B).

- **David Camilo Garcia Muñoz**, **Freddy Andres Saavedra Hoyos**, Oscar Mauricio Caicedo Rendón, Felipe Estrada Solano. **P4Tree: Approach based on Random Forest for Elephant Flows Classifying in Programmable Data Plane**. Elseiver - Journal of Network and Computer Applications.
    - Estado: Escrito y enviado.
    - Clasificación: A1
    - Factor de Impacto: 7.574
    - Índice H: 115 Scimago

1.3. Organización

Este documento de trabajo de grado se ha dividido en los capítulos que se describen a continuación.

- Este capítulo introductorio presenta el planteamiento del problema, delinea la hipótesis, expone los objetivos, resume las contribuciones y describe la estructura general de esta disertación.
- **Capítulo 2** presenta los conceptos fundamentales de la investigación
- **Capítulo 3** revisa las principales investigaciones relacionadas con la clasificación de flujos elefante y ratón en los diferentes planos SDN.
- **Capítulo 4** detalla el mecanismo para clasificar flujos elefante y ratón en Planos de Datos programables, su arquitectura y el algoritmo que describe su funcionamiento.
- **Capítulo 5** presenta la evaluación del mecanismo para clasificar flujos mediante un análisis de desempeño, además, presenta un análisis comparativo con los enfoques más representativos en la literatura.
- **Capítulo 6** presenta las conclusiones y los trabajos futuros.

Capítulo 2

Conceptos Fundamentales

Este capítulo presenta los antecedentes de los principales temas de investigación abarcados en este trabajo de grado. De esta manera, la primera sección contextualiza los conceptos de DCN, SDN y TE. La segunda sección proporciona una introducción a la clasificación de flujos, centrándose en los elefantes y ratones. La tercera sección presenta una descripción de la arquitectura de *switches* programables PISA, seguida de una explicación detallada del lenguaje de programación de Planos de Datos P4, y finalmente una introducción a ML en Planos de Datos programables con énfasis en RF.

2.1. Red de Centro de Datos

Un centro de datos es un conjunto de recursos computacionales y de almacenamiento interconectados mediante una red especialmente diseñada, llamada DCN [46]. DCN asegura el intercambio de tráfico entre máquinas, servicios e Internet y tiene como objetivo proporcionar una capacidad de ancho de banda significativa para lograr un alto *throughput* y baja latencia [8][46]. Diferentes topologías se han propuesto para cumplir con dichos requisitos de rendimiento. Sin embargo, actualmente la mayoría de DCNs emplean topologías basadas en árboles como *Fat-tree* [47] y *VL2* [48], por ejemplo, Altoona de Facebook [49] y Jupiter de Google [50]. Como se observa en la

Figura 2.1, para ofrecer una gran variedad de rutas a cada par de nodos que deban comunicarse, estas topologías organizan en forma de árbol tres capas de *switches* (*i.e.*, capa de núcleo, agregación y borde) y una capa de servidores, los cuales están conectados a la capa de borde o *switches* ToR (*Top-of-Rack*). Comúnmente en DCN, estas topologías emplean ECMP, WCMP o OSPF como protocolo de enrutamiento [17]. No obstante, estos protocolos no tienen en cuenta las condiciones de la red y las características del tráfico. Lo cual limita su capacidad de abordar tráfico de gran velocidad distribuido en flujos elefante y ratón [16], degradando el rendimiento del DCN.

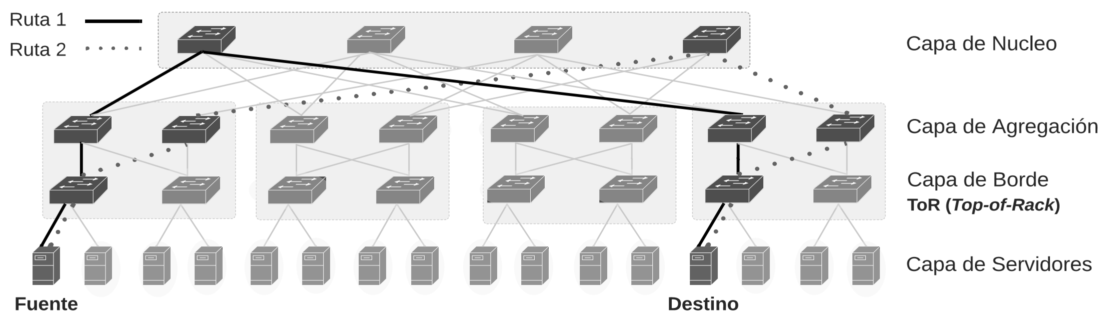

Figura 2.1: Topología de Red: *Fat-tree*

Optimizar los requisitos de rendimiento (*i.e.*, alto ancho de banda, alto *throughput* y baja latencia) sin necesidad de agregar más capacidad a la red es uno de los principales desafíos de TE en DCN. El paradigma SDN representa una gran oportunidad en este ámbito. Ya que mediante su programabilidad y separación del Plano de Control y el Plano de Datos, facilita la implementación de servicios de red (*e.g.*, enrutamiento inteligente [51], balanceo de carga [52] y clasificación de flujos [53]) de manera determinista, dinámica y escalable [54]. Además, SDN brinda flexibilidad, escalabilidad y seguridad al centro de datos, simplifica su administración, reduce los costos y admite tecnología basada en la nube [55]. La arquitectura tradicional de SDN es flexible, lo cual ha permitido realizar mejoras en cuanto las funciones de cada plano. Específicamente, Mestres *et al.* [56] han propuesto emplear un plano SDN (*i.e.*, el Plano de Conocimiento) exclusivamente para transformar información de la red en conocimiento mediante el uso de técnicas ML. Esto con el objetivo de mejorar la gestión y el funcionamiento de la red. El Plano de Conocimiento está

destinado a proporcionar descripciones, recomendaciones y automatización, en pro de diferentes tareas de TE que requieren procesos de toma de decisiones (*e.g.*, enrutamiento inteligente [57]), clasificación (*e.g.*, detección de ataques [58]) y predicción (*e.g.*, predicción de congestión [59]). Para ello aprovecha conjuntamente ML [60], telemetría [61] y análisis de red [62]. Es relevante resaltar que para este trabajo de grado el Plano de Conocimiento es sumamente importante, ya que su poder de cómputo brinda el soporte necesario para entrenar, evaluar y desplegar modelos ML, en los que se fundamenta esta propuesta.

ML es fundamental para TE en DCN. En particular, el enrutamiento inteligente de flujos elefante [22][23] es uno de los métodos que más se ven beneficiados por la inclusión de ML. Ya que este optimiza la función de enrutamiento para minimizar la carga máxima en los enlaces [27] y así evita la congestión (*i.e.*, puntos calientes). Sin embargo, al igual que otros métodos de TE como el enrutamiento inteligente de tráfico sensible a la latencia [24][25] y los balanceadores de carga [26][27], su rendimiento depende de una previa clasificación de los flujos elefante y ratón. Además, dicha clasificación debe ser rápida y precisa, con el objetivo de tomar decisiones lo más pronto posible en pro de la red y hacerlo correctamente.

2.2. Clasificación de Flujos

Un flujo es definido como una secuencia de paquetes que pasa por un punto de observación en la red durante cierto intervalo de tiempo [63]. Todos los paquetes que pertenecen a un flujo en particular tienen un conjunto de propiedades en común, denominado 5-tuplas (*i.e.*, IP de origen y destino, puerto de origen y destino, y protocolo). Además, flujos más complejos consideran campos derivados del tratamiento del paquete y características del paquete en sí.

En general, los flujos se pueden clasificar mediante el uso de umbrales fijos o adaptativos aplicados a diferentes métricas o categorías, tales como duración, tamaño, tasa y ráfagas. Lan y Heidemann [2] definen los tipos de flujos utilizando animales representativos para cada métrica. Por ejemplo, los flujos de larga duración y lentos son denominados tortugas, los flujos elefante son de gran tamaño y larga duración,

a diferencia de los ratones que son pequeños y tardan unos pocos milisegundos. La Tabla 2.1 resume el concepto de cada categoría y su denominación zoológica respecto a un umbral establecido.

Métrica	**Definición**	**Inferior al umbral**	**Superior al umbral**
Duración	Tiempo transcurrido entre el primer y el último paquete del flujo.	Libélulas	Tortugas
Tamaño	Número total de bytes transmitidos en el flujo.	Ratones	Elefantes
Tasa	Número de bytes transmitidos pertenecientes al mismo flujo por unidad de tiempo.	Caracoles	Guepardo
Ráfagas	Espaciamiento entre paquetes pertenecientes al mismo flujo.	Mantarrayas	Puercoespín

Tabla 2.1: Métricas para clasificar flujos según Lan y Heidemann [2]

Los resultados de diferentes estudios relacionados con el comportamiento del flujo revelan que, particularmente en DCN un pequeño porcentaje de los flujos transporta la gran mayoría del tráfico [64][2], estos son los flujos elefante y representan el 10 % del total de flujos, mientras el resto son ratones [65][66]. La clasificación de estos flujos en la práctica está lejos de ser sencilla y un desafío importante es llegar a un consenso respecto al valor de umbral óptimo y estándar que diferencie correctamente un elefante de un ratón [31]. Es común en la literatura encontrar clasificadores que utilizan métodos muy parecidos, pero el valor de umbral difiere totalmente [67][68], esto está ligado directamente al entorno de red donde se realizan las medidas de tráfico que determinan el umbral.

En busca de lograr la clasificación de elefantes y ratones rápida y precisamente, se han explorado algoritmos en el controlador, los servidores y los *switches* PISA de una red SDN. Estos últimos son los primeros en procesar los nuevos paquetes en la red, por lo cual gozan de una ubicación privilegiada en la red y una velocidad de respuesta ideal para realizar la clasificación de flujos a velocidad de línea.

2.3. Plano de Datos Programable

2.3.1. Procesadores de Paquetes Independientes del Protocolo de Programación

La nueva generación de *switches* [69][70] marca una diferencia abismal frente a su antecesora, la distinción radica en una nueva característica incorporada en sus chips: programabilidad. Este tipo de chip se conoce como PISA y es un enfoque novedoso en redes donde el *switch* no ejecuta un código binario incrustado, sino un código interpretado y escrito en un lenguaje de programación de Planos de Datos (*e.g.*, P4 [40] o Lucid [41]). El hardware de *switch* programable hace posible mover la lógica de control al Plano de Datos de la red, lo que mejora el rendimiento de una amplia gama de aplicaciones, por ejemplo, los enrutadores [51], los balanceadores de carga [71][52] y los sistemas de telemetría [53]. Ya que la implementación de la lógica directamente en el Plano de Datos reduce el tiempo de reacción y evita la necesidad de sincronización entre el Plano de Control y el Plano de Datos.

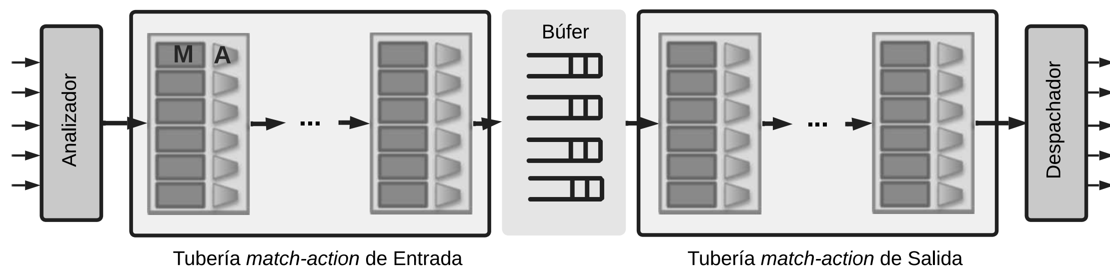

Figura 2.2: Arquitectura de *switch* PISA

Como se muestra en la Figura 2.2, la arquitectura de *switch* PISA consiste en un Analizador programable, una Tubería *match-action* de Entrada, un Búfer, una Tubería *match-action* de Salida y un Despachador programable [41][72]. Además, ambas tuberías cuentan con un conjunto de registros para mantener el estado de las variables a lo largo del tiempo.

Cuando un paquete llega al *switch*, el Analizador programable se encarga de extraer el encabezado. Este módulo permite al programador definir los campos de encabezado que reconoce el *switch*, su tamaño (*i.e.*, el número de bits) y orden, por lo tanto, permite definir los protocolos admitidos por el *switch* [39]. Segundo, la tubería de ingreso, que es una secuencia de tablas *match-action*, determina el puerto de salida y la cola del Búfer en la cual se debe colocar el paquete. Además, en esta tubería el programador define el procesamiento extra que su aplicación requiere. Cada nivel *match-action* (*i.e.*, cada tabla *match-action*), evalúa uno o varios campos de encabezado y realiza una acción, la cual puede ser modificar un campo de encabezado, aplicar una regla de enrutamiento, eliminar el paquete, o incluso modificar registros de memoria para almacenar información en el *switch* y afectar el manejo del tráfico futuro. La ventaja de las tablas *match-action* radica en que permiten reconfigurar sus entradas (*i.e.*, el *match* y la acción) desde el Plano de Control en tiempo de ejecución y sin afectar el funcionamiento del *switch* [73]. Tercero, el paquete ingresa a la tubería de salida, que también consiste en una secuencia de tablas *match-action* para el procesamiento adicional del paquete. Cuarto, finalmente el paquete es enviado a su destino a través del Despachador [74], el cual define como sera el paquete en la salida (*i.e.*, define el formato del encabezado de los paquetes de salida).

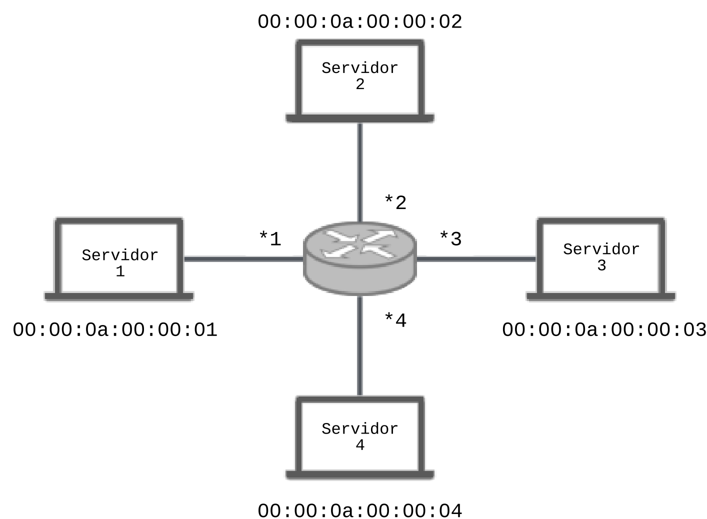

Figura 2.3: Simple topología de un *switch* PISA con 4 servidores conectados

Para esclarecer el concepto de tablas *match-action*, las cuales tienen un rol fundamental en este trabajo de grado, considere el siguiente ejemplo: Se requiere implementar un enrutamiento de capa 2 muy básico que asigne estáticamente el puerto de salida de acuerdo a las direcciones Mac de destino en la topología de la Figura 2.3. Para ello, en la tubería de ingreso, es necesario declarar la tabla *match-action* mostrada en el *Snippet* de codigo 2.1.

```
tabla L2_routing {
  match = {
       dst_Mac: exact;
  }
  actions = {
       SetPort;
  }
}
```

Snippet 2.1: Tabla *match-action* de enrutamiento capa 2

La tabla hace *match* sobre la Mac de destino de los paquetes entrantes, en otras palabras, compara la Mac de destino de los paquetes con todos los posibles valores que puede tomar. Es importante resaltar que todos estos posibles valores deben ser declarados previamente, y se les conoce como entradas de tablas *match-action*. El *Snippet* 2.2, muestra las cuatro entradas *match-action* que es necesario definir para la topología de la Figura 2.3. Cada entrada representa un servidor, y para ello, al lado izquierdo especifica su Mac y al lado derecho especifica tanto la acción que debe realizar la tabla cuando haga *match* con dicha Mac, como los parámetros de que requiere la acción. En este ejemplo, la tabla *match-action* declarada (ver *Snippet* 2.1) únicamente realiza la acción *SetPort* (i.e., asignar puerto de salida). Por lo tanto, al lado derecho las entradas *match-action* especifican el nombre de la acción (*i.e.*, *SetPort*) y como parámetro, especifican el puerto por el cual debe salir el paquete. De esta manera, la tabla *match-action* declarada logra determinar el puerto de salida de los paquetes entrantes dada su dirección Mac de destino.

```
1. Match(00:00:0a:00:00:01) => SetPort(1)
2. Match(00:00:0a:00:00:02) => SetPort(2)
3. Match(00:00:0a:00:00:03) => SetPort(3)
4. Match(00:00:0a:00:00:04) => SetPort(4)
```

Snippet 2.2: Entradas *match-action* que representan cada uno de los servidores de la topología

2.3.2. Programación de Procesadores de Paquetes Independientes del Protocolo

Un Plano de Datos programable implica un nuevo lenguaje de programación y la industria ha colocado sus apuestas en P4 [40], el cual tiene como objetivo cambiar radicalmente la forma como se diseñan los sistemas de red [75]. P4 permite escribir e implementar un programa capaz de especificar el comportamiento de los dispositivos del Plano de Datos, declarar cómo un *switch* PISA procesa los paquetes o especificar una abstracción lógica para el mismo. P4 lleva los beneficios de la ingeniería de software al borde de la red (*i.e.*, redacción de programas, depuración, cobertura de código, etc.). Habilita la posibilidad de agregar procesamiento o análisis de paquetes personalizados al *switch* [75], al igual que implementar rápidamente nuevos protocolos y formatos de encabezado. Además, se crea a partir de 3 principios que le otorgan flexibilidad y escalabilidad [40]:

- Reconfigurabilidad, el controlador puede redefinir el comportamiento de los *switches* sin afectar su funcionalidad.
- Independencia del objetivo, elimina la necesidad de conocer los detalles del *switch* subyacente (*e.g.*, marca, modelo, cantidad de memoria y número de puertos) y permite una programación generalizada.
- Independencia del protocolo, no modela ningún protocolo específico de forma nativa, en cambio, proporciona abstracciones permitiendo al programador expresar los formatos de protocolo existentes y futuros en una sintaxis común [74].

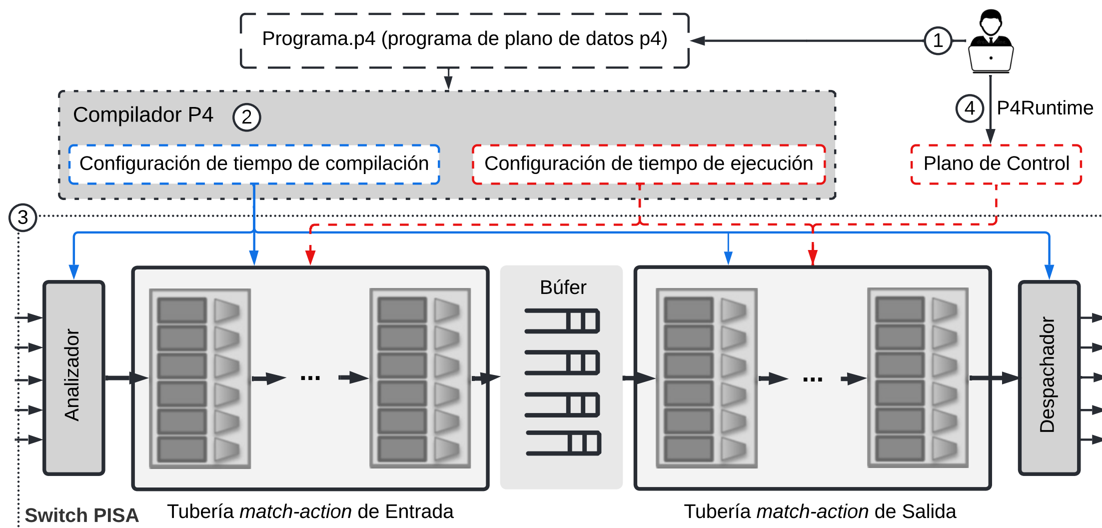

Figura 2.4: Proceso de ejecución de un programa P4 en un *switch* PISA, adaptada de [1]

La Figura 2.4 muestra el proceso de ejecución de un programa P4 en un *switch* PISA. ① El programador desarrolla el programa P4 que describe el comportamiento de uno o varios *switches* PISA. ② El compilador P4 traduce el programa en un archivo JSON y lo envía al *switch*, donde actúa como archivo ejecutable. ③ El *switch* ejecuta el JSON que define el comportamiento del Analizador, Despachador y las tuberías *match-action* de ingreso y salida, con el objetivo de procesar los paquetes de la manera declarada por el programador [76]. ④ El programador controla los elementos del Plano de Datos mediante *P4Runtime* [77] con el objetivo de cambiar el comportamiento del *switch* en tiempo de ejecución. *P4Runtime* es la forma estándar para comunicar el Plano de Control con el Plano de Datos y permite controlar los elementos de un dispositivo o programa definido en P4. Por ejemplo, permite modificar las entradas de las tablas *match-action* de un *switch* o extraer la información almacenada en el *switch* (*i.e.*, los registros y contadores). Para esto, el controlador emplea mensajes Protobuf (*Protocol Buffers*) sobre gRPC (*google Remote Procedure Call*) [78].

El lenguaje P4 aliviana las necesidades de *OpenFlow* como la falta de soporte para más protocolos, aplicaciones y funcionalidades [79]. En la práctica, P4 es fácil de usar y a partir del manejo de metadatos (*i.e.*, variables locales que se eliminan cuando termina el procesamiento del paquete), registros (*i.e.*, memoria para almacenar información a través del tiempo), condicionales (*i.e.*, *if* y *else*), contadores y tablas *match-action*, es posible crear programas complejos para switches PISA (*e.g.*, enrutadores [51], balanceadores de carga [71] o sistemas de telemetría [53]). Además, actualmente el lenguaje cuenta con una serie de primitivas útiles que localizan errores a medida que ocurren en tiempo real [80] y herramientas con las cuales se puede observar el procesamiento de los paquetes dentro del *switch* [81]. Sin embargo, P4 está limitado en cuanto al uso de bucles, números flotantes, multiplicaciones y divisiones, lo cual dificulta la implementación de una gran variedad de programas que requieren su uso (*e.g.*, algoritmos ML).

2.3.3. Machine Learning en Planos de Datos Programables

ML es una rama de la inteligencia artificial que permite a los sistemas aprender automáticamente y proporciona predicciones o soluciones basadas en la experiencia [82]. Además, también se emplea para abordar de manera inteligente problemas complejos de clasificación y optimización. Por estas razones ML ha sido fundamental en las redes de nueva generación [83] y beneficia diferentes enfoques de red que requieren clasificar, predecir o tomar decisiones. Por ejemplo, los algoritmos de enrutamiento inteligente logran predecir o determinar la mejor ruta para el tráfico entrante [57]. Los sistemas de detección de ataques logran clasificar precisamente el tráfico con el objetivo de identificar anomalías, ataques DoS (*Denial-of-Service*) o ataques de fuerza bruta [58]. Y casos de uso específicos como enrutamiento de tráfico sensible a la latencia [25] o balanceo de carga [71], logran clasificar precisamente flujos en elefante y ratón. Sin embargo, hasta hace poco solamente era posible implementar ML en el Plano de Control y en los servidores de la red, ya que estos cuentan con la programabilidad y el poder de cómputo necesario para entrenar y aplicar algoritmos complejos como DNN (*Deep Neural Network*) [84], ANN (*Artificial Neural Network*) [85], RL (*Reinforcement Learning*) [86], DRL (*Deep Reinforcement Learning*) [87] y

AHT (*Adaptive Hoeffding Tree*) [32]. A diferencia del Plano de Datos convencional, que está limitado en cuanto a programabilidad y no permite agregar procesamiento o análisis personalizado de paquetes a los *switches* [39].

La programabilidad de los nuevos *switches* PISA junto a P4, cambian radicalmente el panorama de ML en el Plano de Datos. Recientemente, los trabajos IIsy [42], pHeavy [35], pForest [88] y LSSID [89] han demostrado que es posible entrenar un modelo ML en un servidor o en el controlador de la red, para luego desplegarlo en el Plano de Datos. IIsy implementa los algoritmos SVM (*Support Vector Machine*), KM (*K-Means*), NB (*Naïve Bayes*) y DT (*Decision Tree*) para clasificar tráfico a velocidad de línea en los *switches* y así detectar ataques de red. Con el mismo objetivo, LSSID y pForest despliegan BNN (*Binary Neural Network*) y RF, respectivamente. Por otro lado, pHeavy implementa un DT para clasificar flujos elefante y ratón de manera precisa y a velocidad de línea. Todos estos algoritmos ML implementados en el Plano de Datos tienen una serie de características en común que permiten su despliegue: primero, no emplean bucles para clasificar. Segundo, no emplean multiplicaciones y divisiones, más bien se basan en sumas, restas, negaciones y comparaciones. Tercero, no emplean números flotantes. Y cuarto, son algoritmos de caja blanca [90], es decir, algoritmos que trabajan a partir de funciones matemáticas o lógica que el programador conoce muy bien, por lo tanto, conoce la decisión a la que ha llegado el algoritmo y el proceso para llegar a ella.

Para aplicar el algoritmo ML en el Plano de Datos, los trabajos IIsy, pHeavy, pForest y LSSID siguen la secuencia de acciones detallada en la Figura 2.5. ① A partir de un *dataset* se entrena el modelo ML en el controlador o en un servidor, con el objetivo de emplear diferentes softwares y librerías que permiten el entrenamiento de modelos. Por ejemplo, pForest y IIsy usan Scikit-Learn [91], pHeavy usa Weka [92] y LSSID usa TensorFlow [93]. ② A partir del modelo entrenado el programador realiza un proceso de ingeniería inversa con el objetivo de representar el modelo en la lógica de programación de P4. Por ejemplo, pHeavy aprovecha funciones de Weka para imprimir gráficamente el DT entrenado, lo cual permite analizar el comportamiento del árbol y representarlo en un programa P4. Por otro lado, IIs utiliza el script de inferencia (*i.e.*, el fragmento de código donde Scikit-Learn aplica el modelo entrenado) escrito en Python para crear su propio script en P4. ③ El programa

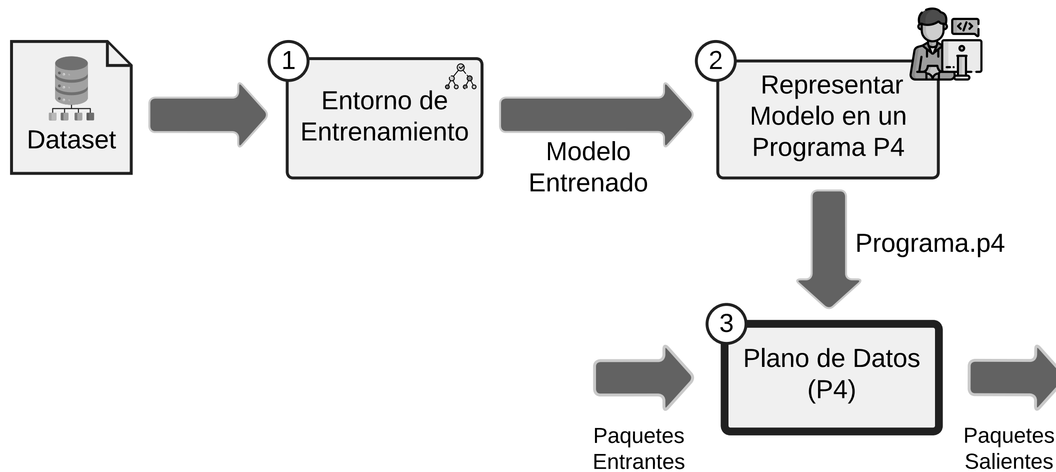

Figura 2.5: Proceso para desplegar ML en el Plano de Datos

P4 es compilado en el Plano de Datos con el objetivo de evaluar su desempeño del modelo con *switches* PISA emulados [94] o *switches* hardware real [95].

La evaluación de rendimiento de los algoritmos ML que hasta ahora se han implementado en el Plano de Datos, concluye que en términos de precisión el algoritmo RF logra los mejores resultados [42][88]. Sin embargo, este algoritmo no ha sido implementado en otros enfoques de red aparte de la clasificación de tráfico para detectar anomalías y ataques DoS. Desaprovechándose así su potencial para mejorar otros enfoques, por ejemplo, el de esta disertación: clasificación de flujos elefante y ratón. Además, cabe señalar que, aunque los autores de pForest y pHeavy comparten información teórica en la literatura sobre RF y DT respectivamente, no comparten el código fuente de su implementación. Finalmente, la viabilidad de implementar RF en el Plano de Datos para clasificar flujos elefante y ratón radica en los siguientes aspectos:

- RF es un algoritmo de caja blanca [90]. Por lo tanto, es posible conocer a detalle como realiza el proceso de inferencia (*i.e.*, como clasifica o predice), incluso es posible representar el modelo gráficamente para después representarlo en P4.
- El proceso de inferencia de un modelo RF se basa en condicionales: Como lo muestra la Figura 2.6, RF es un conjunto de varios DT, donde cada árbol

calcula una etiqueta, y la más repetida será considerada la etiqueta final. Para calcular una etiqueta, en cada nivel del árbol el modelo compara una característica con un umbral especificado por el nodo coincidente. Si la característica supera el umbral, el procesamiento pasa al nodo de la derecha en el siguiente nivel. Si no lo supera, pasa al nodo izquierdo, el cual repite el proceso de comparación de umbrales y así sucesivamente. En el último nivel del árbol, el nodo hoja asigna una etiqueta a la muestra, la cual pasa al proceso de votación que determina la etiqueta final. El proceso de inferencia solo necesita condicionales para calcular las etiquetas y una suma para calcular la etiqueta final [96], permitiendo así su implementación en P4.

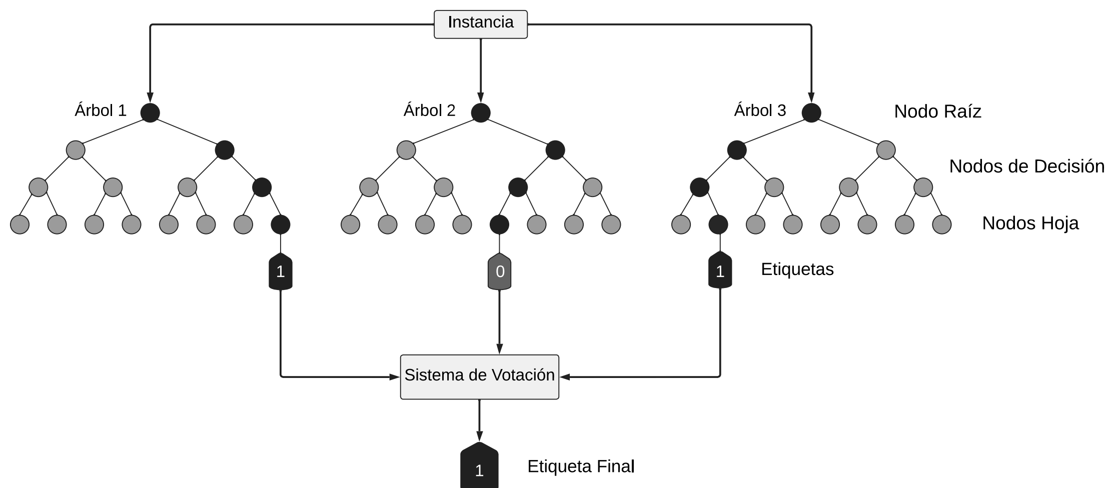

Figura 2.6: Random Forest

- Los condicionales a partir de los cuales el DT calcula una etiqueta se basan en umbrales, los cuales son números flotantes calculados durante el entrenamiento del modelo [96]. Como el Plano de Datos programable no admite el uso de flotantes, es posible aproximarlos a números enteros para adaptar el modelo a las limitaciones del Plano de Datos.

- RF es un algoritmo flexible que permite configurar parámetros de entrenamiento como el número de árboles o la profundidad de cada árbol, lo cual le permite satisfacer diferentes requerimientos de tiempo y precisión.

Nota de autor: *Este capítulo definió los conceptos fundamentales que contextualizan este trabajo de grado. Empezando por los aspectos generales de una DCN, la importancia de TE para optimizar los requisitos de rendimiento (i.e., alto ancho de banda, alto throughput y baja latencia), y como el Plano de Conocimiento favorece diferentes tareas de TE. Luego, el capítulo profundizo en la clasificación de flujos elefante y ratón. Finalmente, realizo la descripción del Plano de Datos programable, definiendo: la arquitectura de switch PISA, el lenguaje de programación de Planos de Datos P4, como este lenguaje ha permitido implementar ML en los switches, y el por qué RF es un algoritmo viable para clasificar flujos elefante y ratón.*

Capítulo 3

Trabajos Relacionados

A continuación, este capítulo presenta la descripción de los trabajos más representativos encontrados en la literatura de clasificación de flujos elefante. Entre los cuales se encuentran mecanismos abordados mediante diferentes modelos y estrategias desde el Plano de Control SDN, los servidores, el Plano de Datos e híbridos.

3.1. Clasificación de Flujos en el Plano de Control

Duque *et al.* [31] implementan un enfoque basado en PSD (*Packet Size Distribution*) y TM (*Template Matching*) para clasificar flujos elefante en el controlador de un DCN. PSD permite capturar el comportamiento y la dinámica de los flujos en forma de plantilla y con la misma precisión que los contadores tradicionales (*i.e.*, duración, paquetes y bytes). Su ventaja radica en que funciona de manera fiable incluso si se dispone de información limitada del flujo. TM se emplea para comparar el flujo con 4 plantillas de referencia, y así determinar su clase. Los autores evalúan la precisión en base a simulaciones y concluyen que utilizar PSD junto a TM brinda una precisión general del 96 %. Sin embargo, este trabajo no considera un tiempo de espera para terminar flujos inactivos (*i.e.*, flujos que no reciben paquetes hace un tiempo), lo cual representa un alto consumo de recursos de almacenamiento. Además, para clasificar

un flujo el controlador debe recolectar 13 paquetes, lo cual genera un alto tiempo de clasificación.

Poupart *et al.* [97] exploran la predicción del tamaño de flujo para clasificar tráfico mediante umbrales fijos. Esta tarea es particularmente desafiante debido a que los patrones del tráfico son cambiantes y el proceso de predicción debe realizarse en unos pocos milisegundos. Ante esta problemática, los autores proponen un sistema centralizado de clasificación que emplea ML en el controlador. Evalúan tres predictores en línea basados en redes neuronales, regresión de procesos gaussiana y coincidencia de momentos bayesianos. Los tres predictores obtienen tiempos de clasificación en la escala de milisegundos, superando al trabajo inmediatamente anterior [31], cuyo tiempo de clasificación ronda los 4 segundos. Sin embargo, a pesar de lograr buenos tiempos de clasificación, la precisión de los predictores no es buena, ya que predicen el tamaño del flujo a partir de 3 paquetes, lo cual no garantiza fiabilidad.

Liu *et al.* [98] plantean que los métodos de detección de flujos elefante basados en umbrales de valor fijo, provocan una gran cantidad de errores (*i.e.*, falsos positivos y falsos negativos: ratones clasificados como elefantes y elefantes clasificados como ratones). Los autores proponen un enfoque de clasificación de flujos adaptativo mediante un algoritmo DTL (*Dynamical Traffic Learning*), el cual es capaz de modificar el valor del umbral a medida que el tráfico varía. Los resultados de la evaluación muestran que el error de clasificación (*i.e.*, falsos positivos y falsos negativos) puede ser reducido hasta el 4.61 % en un escenario de tráfico cambiante. Sin embargo, el controlador sondea los *switches* constantemente para recopilar información del tráfico, lo cual representa sobrecarga de comunicación adicional y afecta el rendimiento del controlador.

Ma *et al.* [99] presentan un enfoque basado AFL (*Asynchronous Federated Learning*) para clasificar flujos elefante de manera distribuida en una red SDN con diversos controladores. AFL permite a cada controlador entrenar un modelo DT diferente en función de las características del tráfico local. Luego, los controladores suben su modelo a la nube de manera asíncrona, para que un modelo global aprenda de toda la red y actualice los modelos locales. Así, cada controlador obtiene una vista generalizada de todo el tráfico de la red y puede clasificar flujos precisamente

aunque el tipo de tráfico sea nuevo. La evaluación emplea 5 trazas de tráfico real y demuestra que la precisión de AFL es superior al 90 % con tráfico que ya ha sido observado y superior al 75 % con tráfico que no se ha observado antes. Además, los autores concluyen que su técnica es más precisa que los principales referentes en clasificación distribuida. Sin embargo, los controladores se ven afectados por el alto consumo de recursos computacionales que genera la técnica AFL.

Los clasificadores de umbral fijo [97][99] se consideran inexactos dada su incapacidad de adaptarse al cambiante comportamiento del tráfico. Las propuestas [31] y [98] solucionan dicha situación adoptando umbrales adaptativos o eliminando la necesidad de usar umbral. Sin embargo, al igual que la mayoría de las propuestas en el Plano de Control, la principal limitación está dada por el alto tiempo que toma recolectar información para clasificar los flujos con precisión. Además, la sobrecarga de procesamiento del controlador y el tráfico generado por la recolección de información en los *switches* son considerablemente altos.

3.2. Clasificación de Flujos en los Servidores

Curtis *et al.* [67] presentan Mahout, un sistema de gestión de tráfico eficaz y de bajo costo en DCN. Este mecanismo sondea los búferes de *socket* en los servidores para detectar, marcar e informar los grandes flujos al controlador de red, el cual los reprograma por diferentes rutas. La velocidad de clasificación en el servidor y la señalización en banda utilizada para el intercambio de información con el controlador, eliminan la necesidad de monitorear el flujo en los *switches* y por lo tanto, genera una baja sobrecarga de comunicación. El tiempo de clasificación ronda unos pocos milisegundos y la sobrecarga de procesamiento afecta el rendimiento sólo cuando el umbral de clasificación supera 1 Gb. A pesar de los buenos resultados de Mahout, es requerida una modificación software en todos los servidores, lo cual limita su escalabilidad a grandes redes.

NELLY [32] es un método eficiente para detectar precisa y oportunamente flujos elefante del lado del servidor en DCN. Aplica una política de clasificación que mejora a través de la experiencia y se basa en identificar el mayor número de flujos elefante,

mientras afecta la menor cantidad de ratones posible, con baja sobrecarga de comunicación y recursos de memoria limitados. El agente de aprendizaje incremental ubicado en el módulo aprendiz, permite a NELLY construir y actualizar constantemente un modelo de clasificación de tamaño de flujo a partir de datos continuos y dinámicos. Dicho modelo es aplicado por el módulo analizador sólo a los flujos potencialmente grandes, evitando así procesar innecesariamente los flujos más pequeños sensibles al retardo. Los autores evalúan el desempeño de 50 algoritmos de aprendizaje incremental, destacando AHOT (*Adaptive Hoeffding Option Tree*), que permite una configuración flexible del umbral, al mismo tiempo proporciona una excelente precisión independientemente del tipo de tráfico. Sin embargo, el enfoque no logra la clasificación a velocidad de línea, así se utilicen los mejores algoritmos. Además, presenta el mismo problema de Mahout: es requerida una modificación software en todos los servidores.

NELLY saca ventaja a Mahout porque hace uso de un módulo basado en ML para ajustar su modelo de clasificación, lo cual repercute en una abstracción y adaptación más precisa respecto al comportamiento del tráfico. No obstante, los enfoques que clasifican flujos en los servidores están limitados principalmente por la necesidad de modificar el software de todos los servidores, lo cual representa problemas de escalabilidad.

3.3. Clasificación de Flujos en el Plano de Datos

CEDRO [100] detecta flujos elefante en el Plano de Datos para gestionar rápidamente escenarios de congestión de enlace. Aprovecha OPP (*Open Packet Processor*) y las funcionalidades de P4 para implementar una estructura basada en módulos directamente en los *switches*. Un primer módulo en la tubería de tablas evita procesamiento innecesario a los flujos más pequeños, marcando como elefante únicamente flujos potencialmente grandes. Otro bloque corrobora la clasificación y si además el flujo en cuestión está experimentando congestión de enlace, notifica a un último módulo que reforma la política de enrutamiento por defecto ECMP y desvía el elefante, eludiendo el camino ya congestionado. Para evaluar el esquema, los autores

observan la respuesta del mecanismo manejando la congestión, en términos del FCT (*Flow Completion Time*), el cual mejora hasta un 5 % en comparación con el obtenido por ECMP. No obstante, el tiempo de clasificación es alto, ya que CEDRO emplea contadores de paquetes para clasificar los flujos. Además, cabe resaltar que la evaluación no considera parámetros importantes como la precisión de clasificación.

HashPipe [101] detecta los k flujos elefante más grandes en el Plano de Datos, con alta precisión y dentro de las restricciones de los *switches* programables emergentes (*i.e.*, memoria y procesamiento). El algoritmo gestiona bien el espacio de almacenamiento debido a que rastrea un número establecido de flujos elefante, proporcional a la memoria limitada del *switch*. Para cada flujo, sólo guarda información de identificación y recuento de paquetes en una tubería de tablas *hash*, que en el caso de saturarse empieza a priorizar los flujos más grandes descartando las claves con menor número de paquetes. HashPipe es evaluado mediante datos etiquetados, los cuales se clasifican con una precisión superior al 90 % tanto en una red convencional como en un DCN. Sin embargo, el enfoque considera que todos los paquetes tienen un tamaño promedio, lo cual puede repercutir negativamente en la precisión si se considera una red con amplia variedad de aplicaciones y tamaños de paquete. Además, si el tráfico aumenta lo suficiente, es posible que circulen más flujos elefante de los que se pueden detectar.

Harrison *et al.* [102] proponen un mecanismo para identificar flujos elefante en el Plano de Datos y con baja sobrecarga de comunicación. Aplican umbrales adaptativos en los *switches* de borde para ajustarse a los cambios de red y cada *switch* comunica al controlador únicamente los grandes flujos que superan el umbral local establecido. Además, el controlador sondea selectivamente los *switches* para obtener recuentos adicionales y actualizar el umbral local. Los autores crean un prototipo utilizando el lenguaje P4 y es simulado con trazas de paquetes del mundo real, como resultado el método presenta una buena precisión de clasificación, con un ahorro aproximado del 70 % en gastos generales de comunicación. Sin embargo, este mecanismo desperdicia recursos de almacenamiento y de procesamiento, dado que carece de una estrategia para no reportar el mismo flujo al controlador desde muchos *switches*. También, se basa en contadores para clasificar, lo cual genera un alto tiempo de clasificación.

Zhang *et al.* [35] presentan pHeavy, un esquema basado en ML para clasificar flujos elefante directamente en el Plano de Datos programable, eliminando así la necesidad de sincronización con el controlador SDN y la sobrecarga de procesamiento. pHeavy entrena un DT en el controlador para luego desplegarlo en el Plano de Datos y así clasificar flujos dentro de la red a velocidad de línea. pHeavy está implementado tanto en el *switch* de software bmv2 como en un *switch* de hardware Barefoot Tofino. Los resultados de la evaluación demuestran que pHeavy logra una precisión del 85 % y el 98 % después de recibir los primeros 5 y 20 paquetes de un flujo, respectivamente. Sin embargo, el DT requiere 20 paquetes para clasificar, lo cual lo hace lento. Además, no puede actualizar el modelo en tiempo de ejecución, lo que limita la flexibilidad ante los cambios de la red.

Kim *et al.* [38] proponen Count-Less, un enfoque basado en contadores para clasificar flujos en el Plano de Datos programable. Count-Less optimiza el uso de los contadores y busca emplear la menor cantidad de memoria posible. Para ello, administra inteligentemente los recursos almacenando varios flujos en un registro. Además, la estrategia inteligente de actualización de contadores le permite a Count-Less procesar rápidamente los paquetes. Para probar la viabilidad de Count-Less clasificando tráfico de alta velocidad, se implementa el prototipo en un *switch* programable basado en P4. Los resultados demuestran que Count-Less es más rápido que los contadores tradicionales procesando paquetes y consume muchos menos recursos computacionales y de almacenamiento. Sin embargo, al estar basado en contadores el *switch* el tiempo de clasificación de elefantes es alto.

La mayoría de los clasificadores en el Plano de Datos se basa en contadores a excepción de pHeavy. Los contadores proporcionan una precisión casi perfecta, sin embargo, el tiempo de clasificación es alto. Además, el despliegue de *switches* programables es un requisito para todas las implementaciones.

3.4. Clasificación de Flujos con Esquemas Híbridos

Huang *et al.* [103] proponen un método de detección de flujos elefante basado en dos clasificadores, uno del lado del *switch* y otro del lado del controlador. El *switch* solo

admite algoritmos ML basados en reglas y árboles de decisión básicos, por lo cual la precisión de clasificación está limitada, dando espacio a errores (falsos positivos y falsos negativos). Los flujos clasificados erróneamente pasan a manos del controlador, que con su gran poder de cómputo clasifica nuevamente el flujo a partir de un algoritmo más complejo, generando mayor precisión y medida-F. Además, el controlador reentrena ambos algoritmos constantemente mejorando la flexibilidad. Los resultados de la simulación muestran que tanto la precisión como la recuperación mejoran a lo largo del tiempo. Además, los autores evalúan diferentes algoritmos de clasificación en el lado del controlador, resultando C4.5 el que brinda mejores resultados. Finalmente comparan el rendimiento con un enfoque de referencia [104], superándolo tanto en tiempo de clasificación como sobrecarga de procesamiento. Sin embargo, la precisión no logra mejores resultados, tampoco se evalúa ni discute la sobrecarga de comunicación (entre el *switch* y el controlador), aspecto muy importante en este tipo de enfoques.

Hamdan *et al.* [43] plantean una técnica de detección de flujos elefante repartiendo la tarea de clasificación entre el *switch* y el controlador. Uno aplica el algoritmo CMS (*Count Min Sketch*) extendido, mientras el otro VFDT (*Very Fast Decision Tree*), respectivamente, con el fin de lograr la clasificación de manera precisa, en tiempo real y con baja repercusión en la sobrecarga de tráfico. Los resultados experimentales muestran que el método de detección de elefantes puede lograr una precisión hasta del 98.13 %, con una mayor tasa de recuperación, medida-F y precisión, comparado con el enfoque [103]. La mayoría de los flujos ratones se pueden filtrar en los *switches*, evitando así enviar grandes cantidades de solicitudes de clasificación al controlador. Sin embargo, el enfoque reporta el mismo flujo al controlador desde varios *switches*, desperdiciando recursos de almacenamiento y de procesamiento, los cuales son limitados en el Plano de Datos.

Liu *et al.* [44] proponen dos métodos de clasificación de flujos. En primer lugar, proponen un enfoque híbrido entre el *switch* y en el controlador basado en aprendizaje residual profundo. El *switch* usa un modelo de preclasificación que se encarga de filtrar una gran cantidad de flujos ratones. El controlador usa un modelo de clasificación exacta, encargado de identificar con precisión los flujos elefante. En segundo lugar, proponen un esquema de 4 clasificadores basados en GRU (*Gated Recurrent*

Unit) para detectar elefante, guepardos, tortugas y puercoespines. La evaluación muestra muy buenos resultados en ambos enfoques en términos de precisión. Además, en comparación con los métodos existentes, se mejoran todas las métricas de precisión. No obstante, el enfoque de clasificación de flujos elefante es lento, ya que requiere de la clasificación exacta del controlador.

Los enfoques híbridos presentan una clasificación precisa, además aprovechan ML en el controlador para ajustar los umbrales. Sin embargo, el despliegue de *switches* programables es requisito para llevar a cabo cualquier implementación y la inclusión del controlador incrementa el tiempo de clasificación.

3.5. Brechas

La Tabla 3.1 resume la configuración del umbral y las deficiencias de las soluciones de clasificación de flujos propuestas en cada región de red. Los métodos que clasifican flujos elefante mediante umbrales fijos son ineficientes, ya que no pueden adaptarse a los constantes cambios del tráfico de red. Las soluciones propuestas en el Plano de Control están limitadas principalmente por el alto tiempo que lleva recolectar la información necesaria de los paquetes para clasificar con buena precisión. Dicha recolección de datos es abordada por los autores mediante extracción periódica de estadísticas de flujo o muestreo periódico, por un lado, la extracción periódica brinda una baja granularidad de los flujos y alta sobrecarga de comunicación entre los *switches* y el controlador, mientras el muestreo, puede perder paquetes de gran tamaño incurriendo en errores de clasificación. Mover la clasificación de flujos a los servidores mitiga los problemas de tiempo y sobrecarga de comunicación, además libera al controlador de procesamiento extra y brinda una precisión (P_C) alta. Sin embargo, es necesaria una modificación software (SW) en todos los servidores, comprometiendo gravemente la escalabilidad a grandes redes.

Los *switches* del Plano de Datos son los primeros en procesar los paquetes que entran a la red, esta característica más su programabilidad los convierte en una buena opción para clasificar flujos. No obstante, no se logra un balance entre el tiempo y la precisión, ya que la mayoría de los trabajos implementados hasta ahora se basan

en contadores, los cuales son muy precisos, pero muy lentos. Además, pHeavy [35], el único trabajo encontrado en la literatura que emplea ML para la clasificación (específicamente DT), también es lento. Todas las implementaciones en el Plano de Datos requieren la renovación del hardware (*HW*) (*i.e.*, la renovación a *switches* PISA). Algunas soluciones híbridas aprovechan el poder superior de cómputo del controlador y la velocidad de respuesta del *switch* para una clasificación eficaz y precisa, aun así, su implementación es compleja e incluir el controlador afecta el tiempo de clasificación.

Obra	**Enfoque**	**Umbral**			**Tc**	**Pc**	**HW**	**SW**
		Características	**Tipo**	**Valor**				
[31]	Controlador	Tamaño y Recuento	Adaptativo	-	Alto	Alta	x	x
[97]	Controlador	Tamaño	Estático	-	Medio	Media	x	x
[98]	Controlador	Tamaño	Adaptativo	-	Alto	Alta	x	x
[99]	Controlador	Tamaño	Estático	10KB	Alto	Alta	x	x
[67]	Servidores	Tamaño	Estático	-	Alto	Perfecta	x	✓
[32]	Servidores	Tamaño	Estático	100KB	Bajo	Alta	x	✓
[100]	P. de Datos	Tamaño	Estático	250KB	Alto	Alta	✓	x
[101]	P. de Datos	Tamaño y Recuento	Estático	-	Alto	Alta	✓	x
[102]	P. de Datos	Tamaño	Adaptativo	-	Medio	Media	✓	x
[35]	P. de Datos	Tamaño	Estático		Medio	Alta	✓	x
[38]	P. de Datos	Tamaño	Estático		Alto	Perfecta	✓	x
[103]	Híbrido	Tasa	Adaptativo	-	Bajo	Media	✓	x
[43]	Híbrido	Tamaño	Estático	-	Medio	Alta	✓	x
[44]	Híbrido	Tamaño y Recuento	Estático	10KB	Bajo	Alta	x	x

Tabla 3.1: Brechas

Nota de autor: *Este capítulo presento la descripción de los trabajos más representativos encontrados en la literatura de clasificación de flujos elefante y ratón. El capítulo describió las ventajas y desventajas de realizar la clasificación en el Plano de Control SDN, los servidores, el Plano de Datos y de manera híbrida. Y luego, resumió las principales brechas de investigación que este trabajo de grado aborda.*

Capítulo 4

Mecanismo basado en Random Forest para Clasificar Flujos Elefante y Ratón en Planos de Datos Programables

Cada día las redes se vuelven exponencialmente más complejas, el gran volumen y variedad del tráfico que circula las DCN y las redes convencionales, representan retos en cuanto al rendimiento de la red y su seguridad. En particular, la clasificación de flujos elefante y ratón representa un papel muy importante para TE. Sin embargo, aún existen retos de investigación para optimizar esta tarea en el Plano de Datos, específicamente en términos de tiempo de clasificación. Este capítulo propone un nuevo método de clasificación de flujos elefante y ratón llamado P4Tree, el cual aplica RF en el Plano de Datos programable de una red SDN para clasificar los flujos de forma rápida y precisa. P4Tree entrena un modelo RF fuera de línea en el Plano de Conocimiento [57]. Ya que a diferencia del Plano de Datos, este cuenta con el poder de computó necesario para realizar el entrenamiento del RF, además, permite el uso de bucles, que son indispensables para entrenar el modelo. Luego, el modelo entrenado es enviado al controlador, que usa P4Runtime [77] para instalarlo en el Plano de Datos programable, específicamente, en los *switches* ToR. Estos *switches* aplican

el modelo RF para clasificar en línea los flujos elefante y finalmente, los reportan al controlador para futuro procesamiento. P4Tree permite actualizar el modelo RF en tiempo de ejecución. Además, su modelo RF es flexible, puesto que permite operar con un modelo de más o menos árboles dependiendo de los requisitos de rendimiento (*i.e.*, precisión y tiempo de clasificación), el desempeño de los *switches* (*i.e.*, la capacidad de procesamiento de paquetes) y la velocidad del tráfico. El rendimiento de P4Tree es evaluado exhaustivamente usando las trazas de tráfico real UNIV1 y UNIV2 [105] y diferentes configuraciones RF en términos de número de árboles. Los resultados de la evaluación muestran que P4Tree hace inferencias precisas con unos pocos paquetes y sirve para cumplir la escala de tiempo de clasificación a velocidad de línea. Por lo tanto, P4Tree informa rápidamente los elefantes al controlador para que sean procesados de forma inteligente lo antes posible. Además, solo necesita estar instalado en una red con *switches* PISA para operar.

4.1. Arquitectura de P4Tree

La figura 4.1 presenta P4Tree, un método de clasificación de flujos que aplica RF en el Plano de Datos programable para clasificar de manera rápida y precisa los flujos elefante. P4Tree ópera en los *switches* ToR de topologías basadas en árboles como *Fat-tree* y puede ser instalado en DCNs o en redes convencionales. P4Tree emplea un modelo RF para la clasificación, ya que es un algoritmo preciso y de caja blanca [90]. Además, este algoritmo se construye a partir de operaciones básicas y condicionales, permitiendo ser mapeado en *switches* programables mediante un lenguaje de programación de Plano de Datos. P4Tree se divide en dos partes, Entrenamiento de P4Tree e *Inferencia de P4Tree*. Primero, Entrenamiento de P4Tree tiene como objetivo entrenar fuera de línea el modelo RF en el Plano de Conocimiento y luego enviarlo al controlador, el cual lo instala dentro de los *switches* ToR. Segundo, *Inferencia de P4Tree* aplica el modelo previamente entrenado para clasificar en línea los flujos en el Plano de Datos. Además, es responsable de informar los flujos clasificados como elefante al controlador para que otros enfoques como el enrutamiento inteligente de flujos elefante [22][23], enrutamiento inteligente de tráfico sensible a la latencia [24][25] y balanceo de carga [26][27], continúen con el procesamiento del

flujo. En aras de la legibilidad, la Tabla 4.1 enumera y describe los símbolos definidos en la arquitectura de P4Tree.

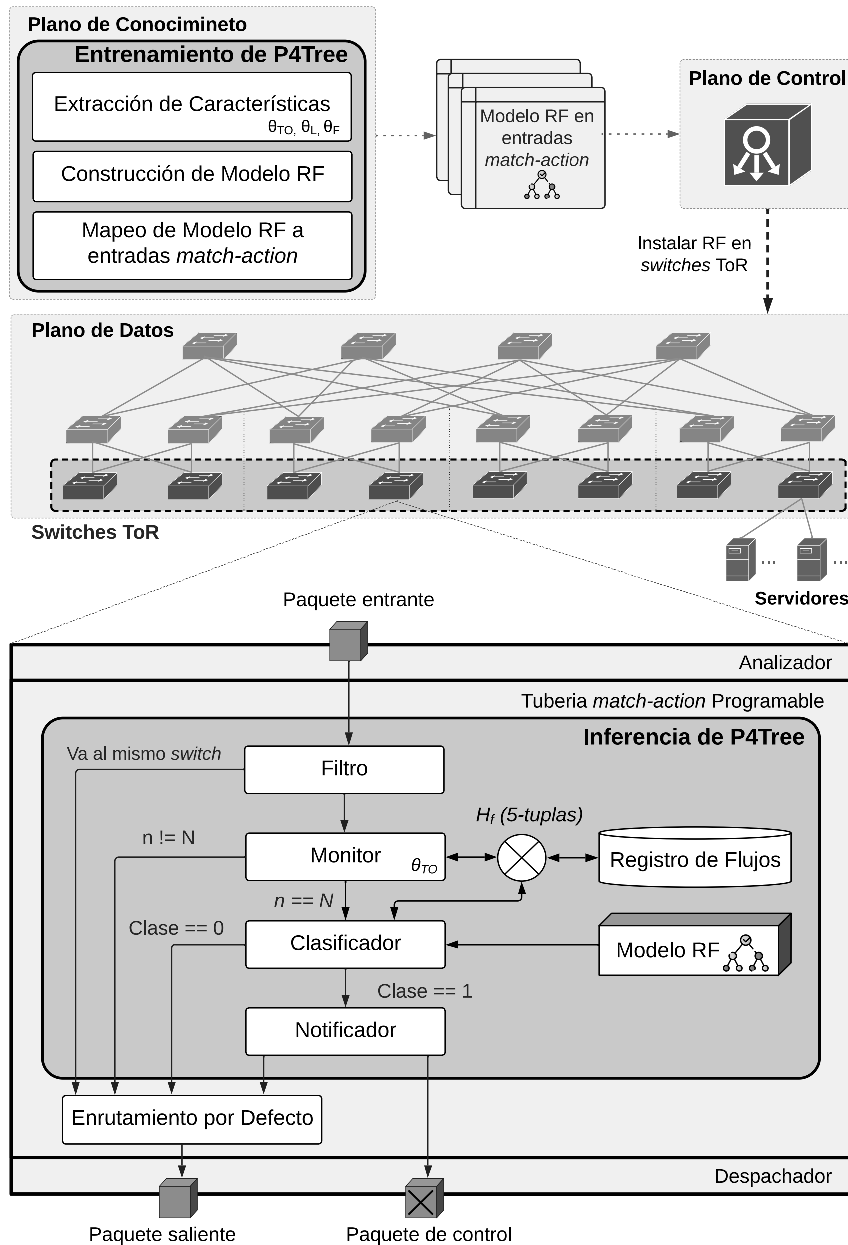

Figura 4.1: Arquitectura de P4Tree

Símbolo	Nombre	Descripción
θ_{TO}	Umbral de tiempo de espera	Límite de tiempo en el que se reconoce que un flujo ha terminado
θ_F	Umbral mínimo de tamaño	Valor mínimo de tamaño de flujo a partir del cual P4Tree considera un flujo para su entrenamiento
N	Número de paquetes	Número de paquetes que se requieren para clasificar un flujo
n	Paquete actual	Número de paquete actual
H_f	Función *Hash*	Función *Hash* que se aplica al encabezado de $5-$ *tuplas* del flujo para calcular una posición de almacenamiento
θ_L	Umbral de etiquetado	Límite de tamaño de flujo que separa los ratones y elefantes

Tabla 4.1: Símbolos en la arquitectura P4Tree

4.2. Entrenamiento de P4Tree

Para entrenar fuera de línea el modelo RF en el Plano de Conocimiento, Entrenamiento de P4Tree divide su operación en tres módulos, Extracción de Características, Construcción del Modelo RF y Mapeo de Modelo RF a entradas *match-action*.

4.2.1. Extracción de Características

Este módulo tiene como objetivo transformar trazas de tráfico real en *datasets* de entrenamiento para el modelo RF de P4Tree. Para ello, recibe trazas de paquetes en formato PCAP como entrada. Luego, las procesa para conformar los flujos a partir del encabezado 5-tuplas (*i.e.*, IP de origen y destino, puerto de origen y destino, y protocolo) y extraer las características de cada uno en un nuevo *dataset* (formato CSV). Las características que definen el *dataset* de entrenamiento de P4Tree respectivamente son: protocolo, puerto de origen, puerto de destino y el peso de los primeros N paquetes de un flujo. Finalmente, este módulo etiqueta el *dataset* con unos y ceros para diferenciar los flujos elefante y ratón, respectivamente. Para este propósito, se aplica un umbral de clasificación θ_L al peso total del flujo (*i.e.*, si un flujo es mayor que θ_L, es un elefante). Cabe señalar que este módulo considera un

tiempo de espera (θ_{TO}) para terminar flujos que no han recibido paquetes desde hace algún tiempo. Además, para la creación del *dataset*, solo se tienen en cuenta flujos con un peso mayor que θ_F. Esto se hace para evitar entrenar el modelo con flujos que definitivamente son ratones sensibles a la latencia y también para asegurar que cada muestra tenga todas las características necesarias para la clasificación, es decir, asegurar que los pesos de los primeros N paquetes del flujo ya han sido recolectados.

4.2.2. Construcción del Modelo RF

Este módulo tiene como objetivo entrenar un modelo RF a partir del *dataset* creado por el módulo de Extracción de Características. Para ello, emplea un ambiente de entrenamiento [91] que permite construir modelos RF y configurar los parámetros de entrenamiento. P4Tree únicamente configura los parámetros que tienen un mayor impacto en el desempeño del modelo RF en términos de precisión y tiempo de clasificación, estos son: el número de árboles, la profundidad máxima de los árboles, el número máximo de nodos de hoja y el peso asociado con la clase negativa y positiva (*i.e.*, cero y uno). Cabe aclarar que el peso asociado con las clases representa la importancia que el modelo RF le da a cada clase. Por lo tanto, el modelo aprende más sobre la clase con un mayor peso. Inicialmente, este módulo entrena el modelo RF con sus parámetros por defecto. Luego, los refina mediante un tuneo hasta que obtiene la configuración más equilibrada, *i.e.*, cuando el RF clasifica con precisión tanto los elefantes como los ratones y tarda menos de $40\mu s$ en etiquetar un flujo como elefante o ratón.

4.2.3. Mapeo de Modelo RF a entradas *match-action*

Al igual que [88], el modelo RF de P4Tree funciona a partir de dos componentes: *código de programa*, que se ejecuta en *switches* PISA, y *configuración de programa*, que especifica el comportamiento de este programa. La diferencia entre el código y la configuración, es que los *switches* PISA permiten cambiar la configuración en tiempo de ejecución, mientras que cambiar el código requiere reiniciar el *switch*. Por

esta razón, P4Tree solo codifica la estructura generalizada del modelo RF y mantiene reconfigurables los parámetros que definen el comportamiento de un modelo RF específico. Cambiar la configuración de los programas compilados en los *switches* PISA se puede hacer de dos maneras: modificando registros o modificando entradas en tablas *match-action*. En este contexto, este módulo mapea el modelo RF entrenado en entradas *match-action*, que configuran el modelo RF del programa compilado en los *switches* ToR (*i.e.*, *Inferencia de P4Tree*).

Para codificar un modelo RF en un *switch*, es necesario definir la estructura y la semántica de RF en el modelo de procesamiento de paquetes de los *switches* PISA. En estos *switches*, toda la lógica de procesamiento de paquetes se define en una tubería programable, que en sí, es una secuencia de tablas *match-action* que permiten actualizar sus entradas desde el Plano de Control [40]. El algoritmo RF es un conjunto de diferentes DTs, y cada árbol es un conjunto de nodos de decisión organizados de forma descendente en niveles [96]. Por lo tanto, cada nivel del árbol es codificado en una tabla *match-action* de la tubería programable, y en cada nivel, un nodo de decisión compara una característica con un umbral especificado. Si el umbral es superado, el procesamiento pasa al nodo derecho en el siguiente nivel (*i.e.*, en la siguiente tabla). En caso contrario, pasa al nodo izquierdo, donde se repite el proceso de comparación de umbrales, y así sucesivamente. Luego, en el último nivel del árbol, el nodo de hoja asigna una etiqueta a la muestra. Como RF consta de un grupo de n diferentes DTs, n etiquetas son calculadas, y la más repetida es considerada como la decisión final. El *Snippet* 4.1 presenta la estructura de una tabla en el enésimo nivel de un DT.

```
tabla nivel_n {
  match = {
      Nodo_anterior: exact;
      Caracteristica_anterior: exact;
      Supero_umbral: exact;
  }
    acciones = {
      CheckFeature;
      SetClass;
  }
}
```

Snippet 4.1: Tabla *match-action* que codifica un nodo en el enésimo nivel de un árbol del RF

El *match* (*i.e.*, las condiciones) utilizado en las tablas para representar los nodos del DT está definido por: el identificador único del nodo anterior, el identificador de la función evaluada en el nodo anterior y el resultado anterior (*i.e.*, si la condición se cumplió o no). El *match* se basa en los valores anteriores para identificar la posición exacta del nodo actual en la estructura de árbol y qué acción específica realiza este nodo. Los nodos de decisión realizan *CheckFeature*, la cual compara una característica con un umbral especificado. Mientras que los nodos hoja realizan SetClass, la cual asigna una etiqueta a la muestra (*es decir*, uno o cero).

```
1. match(Nodo_anterior, Caracteristica_anterior, Supero_umbral) =>
                                        CheckFeature(Caracteristica, Umbral)

2. match(Nodo_anterior, Caracteristica_anterior, Supero_umbral) =>
                                                          SetClass(Clase)
```

Snippet 4.2: Nodo de decisión (línea 1) y nodo de hoja (línea 2) en la semántica de las entradas *match-action*

Tenga en cuenta que este módulo completará el contenido exacto de las tablas. Para esto, utiliza el script proporcionado por [106] para mapear cada nodo del modelo RF entrenado en una entrada *match-action* (ver *Snippet* 4.2). Cada entrada define los valores exactos del *match* (lado izquierdo) y los parámetros de entrada de su acción (lado derecho), *i.e.*, la característica y el umbral para los nodos de decisión, y la etiqueta para los nodos hoja. Una vez que este módulo ha mapeado el modelo RF a entradas *match-action*, las envía al controlador, el cual las instala en los *switches* ToR y permite que el modelo RF del programa compilado (*i.e.*, P4Tree Inference) clasifique.

4.3. Inferencia de P4Tree

Inferencia de P4Tree aplica el modelo RF instalado en los *switches* ToR para clasificar en línea los flujos elefante y luego informarlos al controlador. Como se ilustra en la Figura 4.1, *Inferencia de P4Tree* realiza la clasificación a partir de cuatro módulos ubicados en la tubería programable *match-action* del *switch*: Filtro, Monitor, Clasificador y Notificador. En el algoritmo 1 se detalla el proceso de cada módulo.

Cuando un paquete llega al *switch*, el analizador extrae el encabezado del paquete. El analizador declarado para P4Tree esta habilitado para reconocer únicamente encabezados IPv4, ya que no hay públicamente disponibles trazas de tráfico ni *datasets* IPv6 que permitan entrenar P4Tree y evaluar su rendimiento. No obstante, para escalar P4Tree a IPv6, solamente es necesario declarar su encabezado en el Analizador (*i.e.*, declarar el orden de los campos de encabezado definidos por IPv6 y su tamaño). Una vez el analizador ha extraído los encabezados, la tubería *match-action* programable los usa para procesar el paquete de acuerdo con el programa compilado en el *switch*, *i.e.*, *Inferencia de P4Tree*. El cual comienza el procesamiento del paquete con el módulo Filtro.

4.3.1. Filtro

El Filtro (Algoritmo 1, líneas 1-5) tiene como objetivo evitar el procesamiento adicional de todos aquellos paquetes que se dirigen a un servidor en el mismo *switch*, evitar almacenar información de estos paquetes dentro del *switch*, y además, evitar clasificar y reportar el mismo flujo desde varios *switches*. Para ello, el Filtro compara la IP de destino del paquete con las IP de las interfaces del *switch*. Las IP de las interfaces están almacenadas en un registro, con el objetivo de poder ser definidas por el controlador, el cual conoce la topología de la red, y por ende, cuáles IP se encuentran conectadas al *switch*. Por un lado, si el destino del paquete está conectado directamente (*i.e.*, no debe saltar a otro *switch* para llegar a su destino), el módulo Enrutamiento por Defecto aplica inmediatamente el enrutamiento preconfigurado en la red (*i.e.*, especifica a través de qué puerto del *switch* debe salir el paquete) y coloca el paquete rápidamente en la cola del búfer para que sea despachado a su destino. Esto con el objetivo de que el paquete no sea procesado por los demás módulos de *Inferencia de P4Tree*, los cuales consumen recursos de procesamiento y almacenamiento extra. Por otro lado, si el destino del paquete es un servidor en otro *switch*, el módulo Monitor continúa procesando el paquete.

4.3.2. Monitor

El Monitor (Algoritmo 1, líneas 6-17) tiene como objetivo administrar el *Registro de Flujos*, que es la base de datos donde se almacena la información de los flujos que atraviesan el *switch*. P4Tree almacena cada flujo observado en una posición del *Registro de Flujos*, y en cada posición se almacenan: el encabezado 5-tuplas, el peso de los primeros N paquetes, la hora del último paquete visto y la clase del flujo, que por defecto es ratón. En la línea 6, el Monitor comienza a rastrear los flujos desde el encabezado de 5-tuplas, el tamaño y la marca de tiempo de cada paquete entrante. En la línea 7, el Monitor aplica una función *Hash* sobre el encabezado de 5-tuplas para generar la posición donde se almacenará el flujo. El Plano de Datos programable proporciona diferentes algoritmos *Hash* que pueden ser usados para transformar las 5-tuplas en un nuevo valor de longitud fija, por ejemplo, crc32, crc16, cksum16 y xor16 [107]. P4Tree emplea el algoritmo crc32, ya que es el *Hash* que obtiene mejor desempeño dado su mayor número de bits. En las líneas 8 a 18 el Monitor verifica si la posición calculada por el *Hash* está vacía en el *Registro de Flujos*. Si es así, el Monitor inicia un nuevo flujo (Líneas 33-40). De lo contrario, si ya hay un flujo en esta posición, pueden pasar dos cosas. Primero, que el tiempo transcurrido desde la llegada del último paquete del flujo sea mayor a θ_{TO}. En este caso el Monitor sobrescribe el nuevo flujo en dicha posición (Algoritmo 1, líneas 11-13), con el objetivo de terminar flujos inactivos y optimizar el uso de memoria. Segundo, que el tiempo transcurrido desde la llegada del último paquete del flujo sea menor a θ_{TO}. En este caso el Monitor actualiza el flujo (líneas 41-50), es decir, actualiza la hora de llegada del último paquete del flujo y el peso de los primeros N paquetes.

P4Tree evita clasificar una gran cantidad de flujos ratón (generalmente sensibles a la latencia), ya que el siguiente modulo (*i.e.*, el Clasificador) solo se aplica cuando llega el paquete N. Antes de su llegada, no es necesario clasificar el flujo y después, solo es necesario almacenar el flujo para no reclasificarlo. Por lo tanto, para todos los paquetes diferentes al N, el Monitor aplica directamente el módulo Enrutamiento por Defecto.

4.3.3. Clasificador

El módulo Clasificador (Algoritmo 1, Líneas 18 - 30) tiene como objetivo clasificar los flujos a partir del modelo RF previamente instalado por Entrenamiento de P4Tree en el *switch*. Cuando llega el paquete N de un flujo, el Clasificador extrae del *Registro de Flujos* las características necesarias para la clasificación (*i.e.*, protocolo, puerto de origen, puerto de destino y el peso de los primeros N paquetes de un flujo). Luego, interpreta el modelo RF en forma de entradas *match-action* instalado previamente (ver subsección 4.2.3) para etiquetar el flujo. Y finalmente, actualiza la clase calculada en *Registro de Flujos*. Si el Clasificador etiqueta un flujo como ratón, entonces el módulo *Enrutamiento por Defecto* aplica inmediatamente el enrutamiento preconfigurado en la red y el paquete sale a través del puerto correspondiente. De lo contrario, si el flujo es un elefante, entonces el siguiente módulo (*i.e.*, el Notificador) se encarga de reportarlo al controlador. Cabe señalar que P4Tree permite la reconfiguración de las entradas *match-action* en tiempo de ejecución, de modo que el modelo RF puede actualizarse si es necesario, es decir, si cambian las características del tráfico en la red y el umbral θ_L debe ser actualizado. Para ello, Entrenamiento de P4Tree debe volver a construir el modelo fuera de línea e instalarlo nuevamente en el *switch*.

4.3.4. Notificador

El Notificador (Algoritmo 1, Líneas 18 - 30) tiene como objetivo informar los flujos etiquetados como elefante al controlador. Para informar estos flujos, crea un paquete de control que incluye el encabezado de 5-tuplas del flujo (*i.e.*, IP de origen y destino, puerto de origen y destino, y protocolo) y luego lo enruta hacia controlador. Donde otros enfoques de red pueden aprovechar esta información para tomar decisiones pertinentes. Por ejemplo, los mecanismos de enrutamiento inteligente de flujos elefante pueden manejar especialmente los elefantes para evitar que se congestionen los enlaces, los balanceadores de carga [52] pueden equilibrar precisamente los enlaces o los enrutadores de tráfico sensible a la latencia [25] pueden crear reglas de enrutamiento que favorezcan a los ratones.

Nota de autor: *Este capítulo presento P4Tree, el mecanismo de clasificación de flujos propuesto en este trabajo de grado. Inicialmente, el capítulo planteo la motivación a partir de la cual fue diseñado el mecanismo. Luego, describió de manera general el funcionamiento de P4Tree y presento su arquitectura. Finalmente, detallo Entrenamiento de P4Tree e Inferencia de P4Tree, describiendo cada uno de sus módulos a partir del algoritmo del mecanismo.*

Algoritmo 1: Inferencia de P4Tree

input : encabezado del paquete entrante h_p, tamaño del paquete s_p, marca de tiempo t_p y modelo de clasificación de tamaño de flujo m
output: paquete p y el paquete de control p_c para informar flujos elefante
data : umbral de tiempo de espera de flujo θ_{TO}, número de paquetes necesarios para clasificar N y función hash H_f

```
1  begin proceso de h_p, s_p, t_p
       // Filtro
2      int[] ← vector de interfaces del switch;
3      if h_p.ip_dst ∈ int[] then
4          f ← call Enrutamiento_por_Defecto(p);
5      else
           // Monitor
6          get 5-tuplas de h_p
7          fidx ← calcular posición a partir de 5-tuplas y H_f;
8          if Registro_de_Flujos[fidx] is ∅ then
9              f ← call Iniciar_Flujo(fidx, 5-tuplas, t_p, s_p)
10         else
11             F ← fetch último flujo f ∈ Registro_de_Flujos tal que f.idx = fidx;
12             if (TiempoActual − f.UtimoTiempoVisto) > θ_TO then
13                 f ← call Iniciar_Flujo(fidx, 5-tuplas, t_p, s_p)
14             else
15                 f ← call Actualizar_Flujo(f, fidx, t_p,s_p)
16             end
17         end
           // Clasificador
18         n ← número de paquetes actual en la posición fidx;
19         caractersticas ← get características del flujo del Registro_de_Flujos;
20         if n == N then
21             f.clase ← m.Clasificador(caractersticas);
22             update f → Registro_de_Flujos;
23         end
24         clase ← clase de flujo actual;
25         if clase == 1 then
               // Notificador
26             p_c ← crea paquete de control a partir de las 5-tuplas;
27             f ← call Notificador(p_c);
28             f ← call Enrutamiento_por_Defecto(p);
29         else
30             f ← call Enrutamiento_por_Defecto(p);
31         end
32     end
33 end
34 function Iniciar_Flujo(fidx, 5-tuplas, t_p, s_p):
35     f ← inicializar un nuevo flujo en la posición fidx;
36     f.5tuplas[ ] ← matriz de campos de encabezado de flujo de las 5-tuplas;
37     f.TiempoInicio ← f.UtimoTiempoVisto ← t_p;
38     f.Peso ← f.pesoPaquetes[0] ← s_p;
39     create f → Registro_de_Flujos;
40     return f
41 end
42 function Actualizar_Flujo(f, fidx, s_p, t_p):
43     n ← número actual de paquetes en la posición fidx;
44     if n ≤ N then
45         f.pesoPaquetes[n] ← s_p;
46     end
47     f.Peso ← f.size + s_p;
48     f.UtimoTiempoVisto ← t_p;
49     update f → Registro_de_Flujos;
50     return f
51 end
```

Capítulo 5

Evaluación y Análisis de Resultados

Esta sección presenta la evaluación de P4Tree. La sección 5.1 muestra el prototipo de P4Tree, mientras que las secciones 5.2 y 5.3 presentan los *datasets* y las métricas de rendimiento, respectivamente. La sección 5.4 muestra la configuración del experimento y la secciones 5.5, 5.6 y 5.7 analizan los resultados de la evaluación. La sección 5.8 presenta las observaciones de la evaluación cuantitativa. Finalmente, la sección 5.9 presenta un análisis comparativo con los enfoques de clasificación de flujos elefante y ratón mas representativos en la literatura.

5.1. Prototipo

La Figura 5.1 presenta el prototipo de P4Tree. El Plano del Conocimiento donde opera *Entrenamiento de P4Tree* fuera de línea, está completamente desarrollado en Python 2.7 con las bibliotecas Scapy 2.4.3 [108], Pandas 1.3.4 [109] y Scikit-Learn 0.24.2 [91]. La biblioteca Scapy es una poderosa herramienta interactiva de manipulación de paquetes de red. Pandas es una librería fácil de usar especializada en la manipulación y análisis de estructuras de datos para Python. Scikit-Learn es una biblioteca de ML de código abierto que proporciona herramientas simples y eficientes para el preprocesamiento de datos, entrenamiento de modelos, evaluación de modelos, exportación de modelos y más. El Plano de Datos donde opera *Inferencia*

de P4Tree en línea, está programado en $P4_{16}$ [40] con el objetivo de que P4Tree pueda ser instalado en cualquier *switch* de arquitectura PISA. En el Plano de Control, se desarrolló un nuevo controlador en Python a partir de los ejemplos en [94]. Además, este controlador se comunica con el plano de datos a través de la API P4Runtime [77], la cual es compatible con Python. El prototipo de P4Tree está disponible en [110].

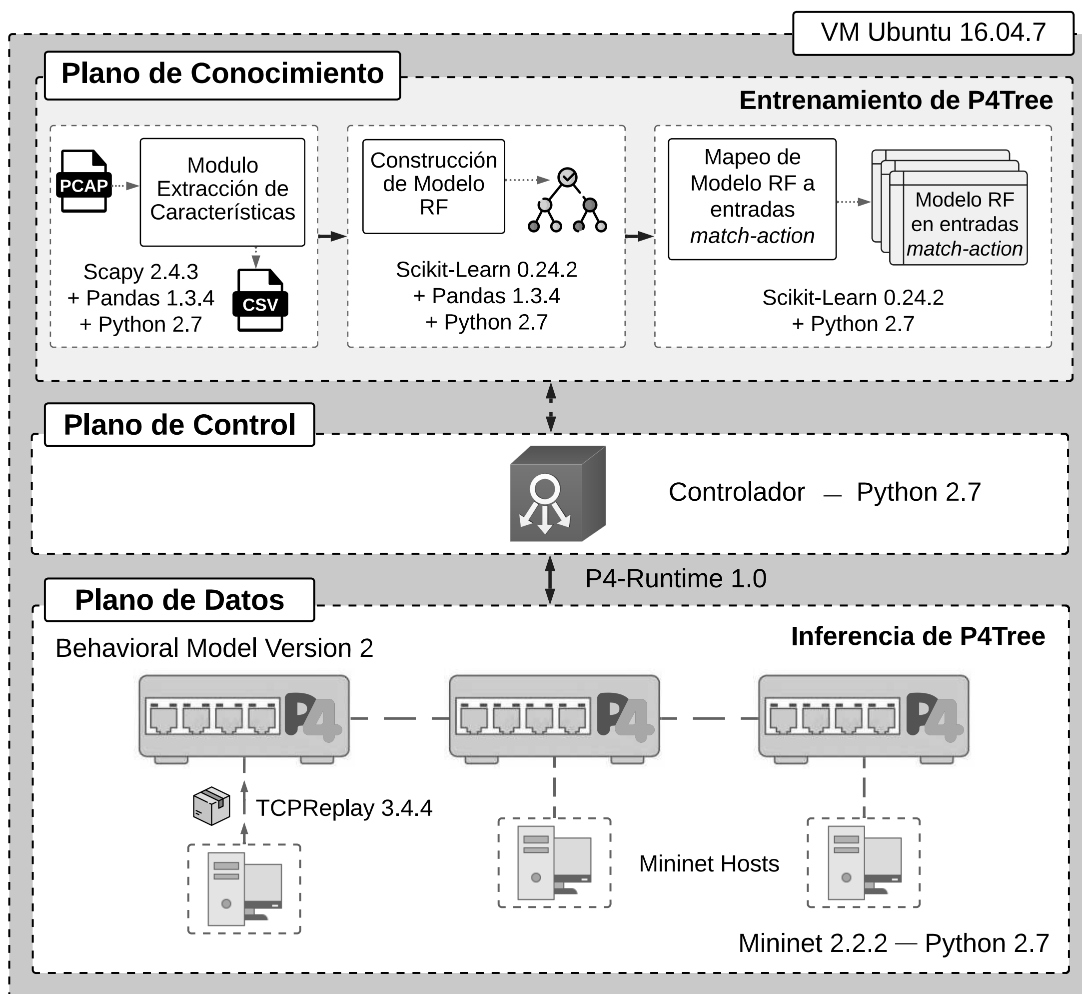

Figura 5.1: Prototipo de P4Tree

5.2. Datasets

Traza de paquetes		**UNIV1**		**UNIV2**	
Duración		3914.62s		9480.35s	
Peso de la traza		12 GB		75 GB	
Paquetes		19,855,388		100,000,000	
Tasa de bytes		3303 kBps		7984 kBps	
Tasa de bits		26 Mbps		63 Mbps	
% IPv4 del total de tráfico		98.98 % *(principalmente TCP)*		99.9 % *(principalmente UDP)*	
Flujos IPv4		1.02 M		1.04 M	
Detalles de flujos IPv4	**Tamaño del flujo**	**% flujos IPv4**	**% tráfico IPv4**	**% flujos IPv4**	**% tráfico IPv4**
	$\geq$ 10 kB	7.16 %	95.06 %	5.91 %	98.81 %
	$\geq$ 100 kB	0.83 %	83.71 %	1.93 %	96.86 %

Tabla 5.1: Detalles de las trazas de tráfico real y los flujos IPV4 obtenidos utilizando el encabezado 5-tuplas y $\theta_{TO} = 5$s

Para el entrenamiento y la evaluación de P4Tree se utilizaron las trazas de tráfico real UNIV1 y UNIV2 [105]. Ambas trazas han sido capturadas en centros de datos universitarios y pertenecen a diferentes servicios, por ejemplo, copias de seguridad de sistemas, alojamiento de sistemas de archivos distribuidos, servidores de correo electrónico y servicios web. El 99 % de los paquetes en las trazas corresponden a tráfico IPv4 e incluyen principalmente flujos TCP y UDP. La Tabla 5.1 resume las características de UNIV1 y UNIV2.

Entrenamiento de P4Tree utiliza las trazas originales en formato PCAP para crear los *datasets* de entrenamiento del modelo RF en formato CSV. Para crear un *dataset*, P4Tree solo necesita definir los parámetros: θ_{TO}, N y θ_L. Se estableció $\theta_{TO} = 5s$ basándose en el al análisis de tiempos de espera en OpenFlow considerado por [111]. Por otro lado, al igual que NELLY [32], P4Tree solo clasificara flujos superiores a $\theta_F = 10kB$ por dos motivos. Primero, por debajo de este valor esta más del 92 % de los flujos (*i.e.*, flujos que definitivamente son ratones sensibles a la latencia). Segundo, por encima de este valor están todos los elefantes potenciales (*i.e.*, flujos que transportan el 95 % del total de tráfico). Como tal, para que un flujo pese más de $10kB$ debe tener un mínimo de N = 7. Los flujos con menos de 7 paquetes por defecto se consideran ratón con el objetivo de evitarles procesamiento innecesario, por lo tanto, se eliminan de los *datasets* de entrenamiento. Se fijó el umbral de

clasificación $\theta_L = 100kB$ para etiquetar los flujos como elefantes o ratones. Este valor fue elegido para comparar el desempeño de P4Tree con otros trabajos, ya que no se ha definido un umbral estándar en la literatura y sigue siendo un desafío de investigación [31]. Como se observa en la tabla 5.1, con $\theta_L = 100kB$ en UNIV1 y UNIV2, los elefantes representan menos del 2 % de los flujos, mientras los ratones representan más del 98 %. Lo cual indica que ambas trazas de tráfico tienen una distribución desequilibrada de clases (*i.e.*, el porcentaje de muestras de una clase es mucho mayor que el porcentaje de la otra clase).

Las trazas originales en formato PCAP también son empleadas para evaluar el desempeño de P4Tree clasificando tráfico a velocidad de línea. Para ello, las trazas son introducidas en la red a mediante Tcpreplay [112] y clasificadas en tiempo real por *Inferencia de P4Tree*. Tcpreplay es un conjunto de utilidades para editar y reproducir tráfico de red capturado previamente en formato PCAP. Además, permite aumentar o disminuir la velocidad original de las trazas de tráfico, establecer un valor de velocidad máxima e incluso controlar la cantidad de paquetes que reproduce en un segundo.

5.3. Métricas de Rendimiento

El rendimiento de P4Tree se evalúa a partir de las métricas de precisión TPR (*True Positive Rate*), FPR (*False Positive Rate*) y MCC (*Matthews Correlation Coefficient*), el tiempo de etiquetado (T_E) y los Paquetes Eliminados. Estas métricas son empleadas en la literatura para evaluar clasificadores de elefantes [32] y clasificadores en el plano de datos [106].

5.3.1. Métricas de Precisión

TPR, FPR y MCC son métricas derivadas de la matriz de confusión, ilustrada en la Figura 5.2. Cada fila en la matriz de confusión representa un resultado predicho y cada columna representa la instancia real. Así, TP (*True Positives*) y TN (*True negatives*) representan los resultados bien predichos para las instancias positivas y

Instancia Real

Resultado Predicho	Positivo (P)	Negativo (N)
P	Verdadero Positivo (TP)	Falso Positivo (FP)
N	Falso Negativo (FN)	Verdadero Negativo (TN)

Figura 5.2: Matriz de Confusión

negativas reales, respectivamente. Por otro lado, FP (*False Positives*) y FN (*False Negatives*) representan los resultados mal predichos para las instancias positivas y negativas reales, respectivamente. Teniendo en cuenta que P4Tree tiene como objetivo identificar tantos elefantes como sea posible, afectando a la menor cantidad de ratones sensibles a la latencia. Los flujos elefante son considerados como la condición positiva. A partir de esto, TPR especifica el porcentaje de elefantes bien clasificados, mientras que el FPR describe el porcentaje de ratones mal clasificados. Las ecuaciones 5.1 y 5.2 describen el cálculo de TPR y FPR a partir de los parámetros de la matriz de confusión.

$$TPR = \frac{TP}{TP + FN} \tag{5.1}$$

$$FPR = \frac{FP}{FP + TN} \tag{5.2}$$

El MCC se utiliza para analizar el equilibrio entre TPR y FPR. Para este propósito,

proporciona una medida entre -1 y 1 a partir de la matriz de confusión, que solo se aproxima a 1 cuando el algoritmo clasifica bien tanto a los elefantes como a los ratones (*i.e.*, cuando se maximiza el TPR y se minimiza el FPR). Si el MCC está por debajo de 0, significa que el algoritmo es menos preciso que un clasificador aleatorio. La ecuación 5.3 describe el cálculo de MCC a partir de los parámetros de la matriz de confusión.

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5.3}$$

5.3.2. Tiempo de Etiquetado

El tiempo de clasificación T_C representa el período de tiempo desde que llega el primer paquete de un flujo hasta que P4Tree lo etiqueta como elefante o ratón, y está descrito en la ecuación 5.4 Donde, T_R especifica el tiempo de recolección de características del flujo y depende directamente del tiempo que tardan en llegar los primeros 7 paquetes. Por otro lado, T_E representa el tiempo que toma el RF de P4Tree en procesar las características y etiquetar el flujo. T_E empleo para el análisis de rendimiento de P4Tree como [32] (ver sección 5.7), mientras T_C se empleó para el análisis comparativo de P4Tree (ver sección 5.9).

$$T_C = T_R + T_E \tag{5.4}$$

5.3.3. Paquetes Eliminados

Los Paquetes Eliminados representan el porcentaje de paquetes que el switch descarta cuando su cola de búfer está llena [106]. Este porcentaje depende de la velocidad del tráfico y la capacidad de P4Tree para procesar paquetes individuales. Además, se calcula comparando la cantidad de paquetes que entran al *switch* con la cantidad de paquetes que son procesados exitosamente y salen del *switch* hacia su destino.

5.4. Configuración del Experimento

Se despliega el Plano de Conocimiento, Control y Datos sobre una máquina Ubuntu 16.04.7, con un procesador Core i7-9700, 8 núcleos de CPU y 16 GB de RAM. Se emula una red simple con un *switch* programable y dos hosts al igual que [106] y [38]. Para ello, se emplea Mininet 2.2.2 [113] con el *switch* de software BMV2 (*Behavior Model Version 2*). BMV2 [94] es una poderosa herramienta para desarrollar, probar y depurar Planos de Datos P4 y software de Plano de Control escrito para ellos. El rendimiento de BMV2 es afectado por dos factores. En primer lugar, qué tan rápido BMV2 puede procesar paquetes individuales: el PPT (*Packet Processing Time*) en BMV2 depende del rendimiento del hardware de la máquina (*i.e.*, cuántos núcleos de CPU, cuánto caché de CPU, etc.) y la complejidad del programa P4 que compila el *switch* (*i.e.*, el número de líneas de código). Si el PPT es alto, la cola del búfer fluye más lento y aumenta el riesgo de que el *switch* elimine paquetes. Se debe esperar un mejor rendimiento en un *switch* real [106]. En segundo lugar, la velocidad del tráfico que cruza el *switch*: si la velocidad del tráfico es lo suficientemente alta, la cola del búfer se llena y obliga al *switch* a eliminar paquetes.

Parámetro	Definición	Valor
max_depth	Máximo nivel de profundidad de un árbol	15
max_leaf_nodes	Máximo número de nodos de hoja de un árbol	200
n_estimators	Número de árboles	variable
class_weight [W_1 : W_0]	Pesos asociados a las clases uno y cero	variable

Tabla 5.2: Configuración de entrenamiento para RF en Scikit-Learn

Para evaluar el rendimiento de P4tree, el modelo RF se entrena en el Plano del Conocimiento con la configuración predeterminada de Scikit-Learn para RF, excepto por los parámetros que se detallan en la tabla 5.2. La profundidad máxima y el número máximo de nodos de hoja se fijaron en 15 y 200 respectivamente, con el objetivo de evitar árboles grandes y lentos. El número de árboles varía en los experimentos. El peso W_0 asociado con la clase negativa (*i.e.*, la clase ratón) se mantiene por defecto (*i.e.*, 1) y el peso W_1 asociado con la clase positiva (*i.e.*, la clase elefante) varía en los experimentos. Cabe aclarar que W_0 y W_1 representan la importancia de cada

clase ante el RF [114]. Por lo tanto, el clasificador aprende más sobre la clase con un peso mayor.

5.5. Análisis de Paquetes Eliminados

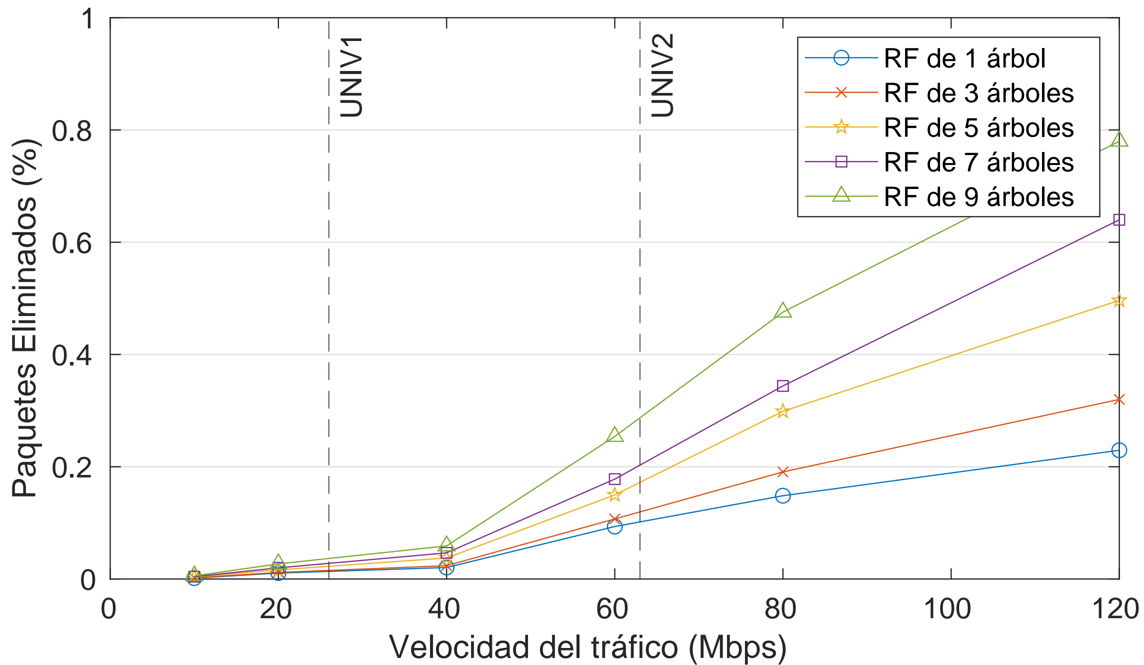

Figura 5.3: Test de Estrés. Se resalta la velocidad original de UNIV1 y UNIV2.

Para evaluar el rendimiento de P4Tree procesando paquetes a velocidad de línea respecto a la cantidad de árboles del modelo RF, se realizó un test de estrés. El cual consiste en enviar las trazas de tráfico originales (*i.e.*, UNIV1 y UNIV2) de un host al otro a través del *switch* y observar el porcentaje de Paquetes Eliminados. Luego, la velocidad original de las trazas se incrementa y se decrementa con el objetivo de encontrar la velocidad máxima a la cual P4Tree presenta un óptimo rendimiento en términos de Paquetes Eliminados. La figura 5.3 muestra el porcentaje de Paquetes Eliminados cuando P4Tree usa un modelo RF con 1, 3, 5, 7 y 9 árboles para clasificar el tráfico a diferentes velocidades. Los resultados muestran que el porcentaje de Paquetes Eliminados aumenta a medida que la velocidad del tráfico aumenta, además, los Paquetes Eliminados también aumentan a medida que aumenta el número de ár-

boles. Como era de esperar, el tráfico de alta velocidad llena rápidamente la cola del búfer y obliga al *switch* a eliminar paquetes. Por otro lado, un RF con más árboles requiere una mayor cantidad de tablas *match-action* para clasificar un flujo. Esto representa un costo de procesamiento más alto y tiene un impacto negativo en el flujo de la cola del búfer, lo que obliga al *switch* a eliminar paquetes. En UNIV1, cuya velocidad promedio es de $26Mbps$, el porcentaje de Paquetes Eliminados asciende hasta el 6 % a medida que aumenta el número de árboles, mientras que los Paquetes Eliminados en UNIV2 ($63Mbps$) llegan al 50 %. Por este motivo y para evaluar las métricas de precisión (*i.e.*, TPR, FPR y MCC) bajo condiciones ideales (*i.e.*, en un escenario donde P4Tree puede procesar correctamente las trazas de tráfico sin Paquetes Eliminados), se debe disminuir la velocidad de UNIV1 y UNIV2 a $10Mbps$ mediante Tcpreplay [112] para garantizar un porcentaje de Paquetes Eliminados casi nulo.

5.6. Análisis de Métricas de Precisión

Algoritmo	**UNIV1** - $W_1 = 6$			**UNIV2** - $W_1 = 2$		
	TPR(%)	**FPR(%)**	**MCC**	**TPR(%)**	**FPR(%)**	**MCC**
RF de 1 árbol	87.1	4	0.36	75.1	5.3	0.39
RF de 3 árboles	88.2	4	0.36	75.2	5.1	0.4
RF de 5 árboles	88.7	3.5	0.38	75.3	5	0.4
RF de 7 árboles	89.5	3.3	0.39	76.2	4.9	0.41
RF de 9 árboles	90.1	3.1	0.4	78.4	4.5	0.43

Tabla 5.3: Métricas de Precisión

La tabla 5.3 presenta el TPR, FPR y MCC cuando P4Tree usa un modelo RF con 1, 3, 5, 7 y 9 árboles para clasificar las trazas UNIV1 y UNIV2. Los valores de la matriz de confusión necesarios para calcular las métricas de precisión se extraen de los registros de *switches* programables mediante P4Runtime. La tabla 5.3 solo incluye resultados que maximizan MCC (*i.e.*, la configuración más equilibrada) y logran un TPR mínimo del 75 %. En UNIV1 el MCC es máximo cuando el modelo RF se entrena con $W_1 = 6$, mientras que en UNIV2 esto se logra con $W_1 = 2$. Los resultados muestran que a medida que aumenta el número de árboles, el TPR, FPR

y MCC mejoran significativamente. En UNIV1 P4Tree con un modelo RF de 1 árbol obtiene una TPR del 87 %, mientras que un modelo RF de 9 árboles obtiene un 90 %. En UNIV2 el TPR varía del 75 % al 78 %. El FPR también mejora cuando el modelo RF es más robusto, en UNIV1 el FPR varía del 4 % al 3 % a medida que aumenta el número de árboles, mientras que en UNIV2 varía del 5,3 % al 4,5 %. Los mejores resultados obtenidos por P4Tree en términos de MCC son 0.4 y 0.43 para UNIV1 y UNIV2, respectivamente. Esto se debe al gran desequilibrio de clases en las trazas de tráfico, que solo se puede mitigar hasta cierto punto.

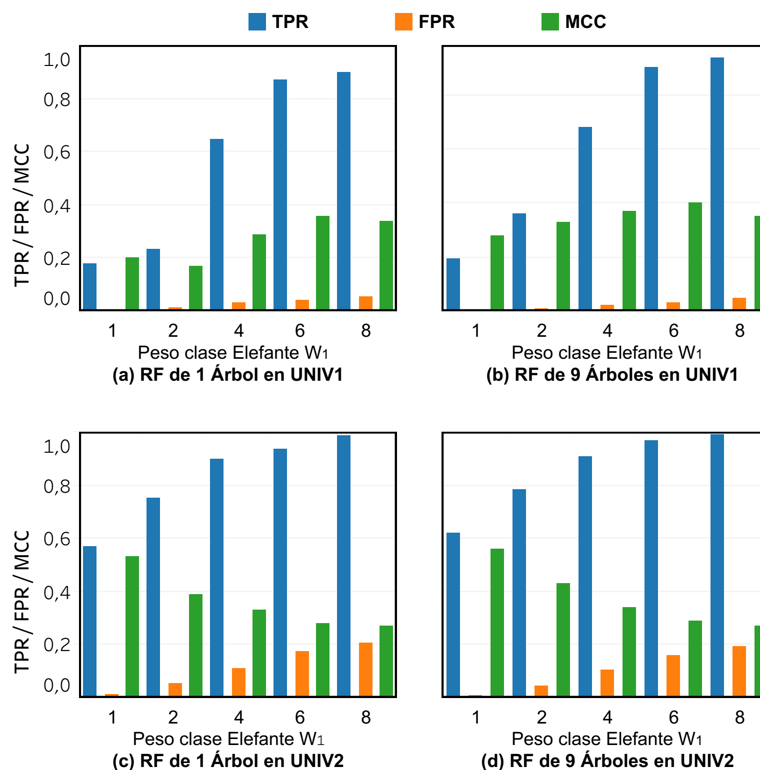

Figura 5.4: Precisión de P4Tree con un modelo RF de 1 árbol (izquierda) y un modelo RF de 9 árboles (derecha) al variar los pesos de los flujos elefante W_1 para UNIV1 (arriba) y UNIV2 (abajo)

Los modelos RF de 1 árbol y RF de 9 árboles se entrenaron manejando diferentes rangos de pesos W_1 para analizar su efecto en la precisión de P4Tree. En la figura 5.4, W_1 varía entre 1 y 8, ya que por encima de este valor el TPR del P4Tree no mejora y el FPR aumenta significativamente. Además, la Figura solo muestra el TPR, FPR

y MCC con W_1 igual a 1,2,4,6 y 8. Los resultados muestran que P4Tree logra un TPR más alto a medida que W_1 aumenta (hasta 95 % en UNIV1 y 99 % en UNIV2). Esto se debe a que establecer mayores pesos para la clase elefante permite ponderar las instancias de datos para aumentar la importancia de clasificar erróneamente los elefantes y disminuir la importancia de clasificar erróneamente los ratones. Lo cual resulta en un clasificador más preciso en términos de flujos elefante a expensas de una ligera degradación del FPR. Además, la compensación entre TPR y FPR (*i.e.*, MCC) mejora para UNIV1 a medida que W_1 aumenta, mientras que para UNIV2 disminuye. Esto se debe a la diferencia entre la distribución de elefantes y ratones en las dos trazas de tráfico (UNIV2 tiene un mayor porcentaje de flujos de elefantes que UNIV1). En conclusión, P4Tree admite una configuración de pesos flexible para cumplir con los diferentes requisitos de TPR y FPR.

5.7. Análisis de Tiempo de Etiquetado

Algoritmo	**UNIV1**	**UNIV2**
	$T_E(\mu\text{s})$	$T_E(\mu\text{s})$
RF de 1 árbol	9.7	14.3
RF de 3 árboles	31.2	42.7
RF de 5 árboles	54.8	68.1
RF de 7 árboles	71.4	87.1
RF de 9 árboles	90.6	103.7

Tabla 5.4: Tiempo de Etiquetado

La tabla 5.4 presenta el T_E cuando P4Tree usa un modelo RF con 1, 3, 5, 7 y 9 árboles para clasificar las trazas UNIV1 y UNIV2. Para obtener el T_E del *switch*, es necesario realizar modificaciones en el código fuente de BMV2 para crear contadores de tiempo en el *switch*, ya que actualmente no cuentan con esta función. Con respecto al T_E en UNIV1, cuando P4Tree usa un modelo RF de 1 árbol, se necesitan 9,7μs para etiquetar el flujo como elefante o ratón. Este tiempo de decisión es de aproximadamente $0,7x$ el PTT en el *switch* emulado (*i.e.*, 14μs) y representa menos de $3,9\,\%$ del RTT (Round-Trip Time) en DCN (*i.e.*, 250μs [115]). Como se muestra

en la tabla 5.4, el T_E aumenta a medida que aumenta la cantidad de árboles, por ejemplo, un modelo RF de 3 árboles requiere 31,2μs, mientras que un modelo RF de 5 árboles toma 54,8μs. Esto se debe a que cada árbol usa una cantidad fija de *match-action* para generar una etiqueta, y a medida que la clasificación requiere más tablas, el T_E aumenta. En UNIV2, cuando P4Tree usa un modelo RF de 1 árbol, obtiene un T_E de $14,3\mu s$, que es $1x$ el PPT y representa $5,7\,\%$ del RTT. Mientras que un clasificador más robusto como el modelo RF de 7 árboles o el modelo RF de 9 árboles requiere $87,1\mu s$ y $103,7\mu s$, respectivamente.

5.8. Observaciones

Para concluir respecto a la configuración que logra el mejor rendimiento en P4Tree, se deben tener en cuenta los siguientes aspectos:

- Hay un equilibrio entre etiquetar los flujos con precisión y hacerlo rápido. Por un lado, si el RTT de la red es estricto, la mejor opción es usar un modelo RF con 3 árboles, ya que T_E está por debajo del 12,5 % del RTT y las métricas de precisión son aceptables. Por otro lado, si el RTT es flexible, es posible usar un RF con 5 árboles para mejorar significativamente las métricas de precisión.
- Los Paquetes Eliminados están directamente relacionados con la velocidad del tráfico y la cantidad de árboles del RF de P4Tree. Sin embargo, la influencia de la cantidad de árboles en los Paquetes Eliminados se vuelve importante a partir de 5 árboles. Por esta razón, se recomienda utilizar un modelo RF con un máximo de 3 árboles. Además, se espera un mejor rendimiento con un *switch* real que tenga la capacidad de procesar más paquetes por segundo (*e.g.*, el *switch* Barefoot Tofino [69]).

5.9. Análisis Comparativo

Esta sección presenta la comparación de P4Tree con los enfoques más representativos encontrados en la literatura de clasificación de flujos elefante: NELLY [32], TTHH [31], Count-Less [38], pHeavy [35] y DL-clasificación2 [44]. NELLY aplica aprendizaje incremental en los servidores para clasificar de manera efectiva los flujos elefante. TTHH utiliza una técnica novedosa basada en PSD y TM para clasificar los flujos en el Plano de Control. En el Plano de Datos programable, Count-Less utiliza contadores de paquetes para clasificar perfectamente todos los flujos, mientras que pHeavy es el primer trabajo hasta donde sabemos que explora ML para clasificar elefantes en el Plano de Datos. Y finalmente, DL-classification2 es un enfoque híbrido que tiene una clasificación previa en el *switch* y una clasificación más precisa en el controlador. Los resultados de UNIV1 se utilizaron para comparar el desempeño de los trabajos en relación con: Técnica de Clasificación, Elefantes Detectados, Elefantes Falsos, Tiempo de Clasificación, Modificaciones de Red y Factores de Desempeño.

Métrica	**Nelly** [32]	**TTHH** [31]	**Count-less** [38]	**pheavy** [35]	**DL-classification2** [44]	**P4Tree**
Técnica de Clasificación	Incremental	TM	Contadores	DT	Aprendizaje Residual Profundo	RF
Elefantes bien Clasificados	95 %	95 %	Perfecto	90 %	93 %	90 %
Elefantes Falsos	2.7 %	2.7 %	Ninguno	3.2 %	5.6 %	3.2 %
Tiempo de Clasificación	0.8s	8.5s	3.8s	2.6s	0.6s	0.8s
Modificaciones de Red	Software en servidores	Ninguno	Hardware de *switches*	Hardware de *switches*	Ninguno	Hardware de *switches*
Factores de Rendimiento	Servidores	Controlador	Switches ToR	*Switches* ToR	Controlador y *switches* ToR	*Switches* ToR

Tabla 5.5: Comparación de P4Tree con trabajos relacionados

5.9.1. Técnica de Clasificación

Los flujos elefante se pueden clasificar a partir de diferentes técnicas según el plano de la red donde se encuentre el clasificador. Tanto en el Plano de Control como en los servidores, es posible implementar algoritmos complejos que requieren todo tipo

de operaciones (*e.g.*, multiplicaciones, divisiones, operaciones con números flotantes, etc.) y bucles para lograr una alta precisión. Por esta razón, NELLY puede implementar algoritmos incrementales como AHT y ARF (*Adaptive Random Forest*), DL-classification2 puede implementar Aprendizaje Residual Profundo y TTHH puede ejecutar su técnica basada en TM.

Por otro lado, el Plano de Datos está limitado en términos de operaciones y bucles, lo que impide la implementación de las anteriores técnicas. La mayoría de los trabajos que clasifican elefantes y ratones en el plano de datos se basan en contadores, a excepción de pHeavy y P4Tree. Los contadores como Count-Less no ignoran ningún flujo, por lo que deben almacenar su información (*i.e.*, el encabezado de 5 tuplas y el conteo de paquetes) hasta que se exceda un umbral establecido, lo que genera una precisión perfecta a expensas de un alto tiempo de clasificación y consumo excesivo de memoria. P4Tree y pHeavy buscan mitigar estos problemas implementando un modelo ML, el cual permite clasificar flujos con los primeros N paquetes mantenido buena precisión.

PHeavy logra implementar un DT dentro del *switch* programable mapeando manualmente un árbol con condicionales, esto lo convierte en un modelo fijo que no se puede actualizar, además su código fuente no está disponible a la fecha. Por otro lado, P4Tree presenta un método para ensamblar varios árboles y formar un algoritmo RF. Este modelo de clasificación es mucho más robusto que pHeavy, logra mejores métricas de precisión y reduce el tiempo de clasificación de los flujos elefante. A diferencia de pHeavy, P4Tree también proporciona un mecanismo que permite actualizar el modelo desde el controlador a través de P4Runtime para adaptarse al tráfico de red cambiante.

5.9.2. Elefantes Detectados

Es importante que los métodos de clasificación de flujo informen tantos elefantes como sea posible. NELLY y TTHH clasifican más de 95 % de flujos elefante, DL-classification2 supera el 93 %, P4Tree y pHeavy logran un TPR superior a 90 % y Count-less proporciona una clasificación perfecta, ya que se basa en contadores de

paquetes. P4Tree puede lograr una precisión de más de 95 % cuando se incrementa el peso de la clase elefante (W_1), sin embargo, el FPR se degrada ligeramente.

5.9.3. Elefantes Falsos

Los ratones que se clasifican erróneamente como flujos elefante (*i.e.*, FPR) se informan al plano de control y se procesan innecesariamente. DL-classification2 obtiene el FPR más alto ($5,6\,\%$), P4Tree y pHeavy clasifican erróneamente menos de $3,2\,\%$ de ratones, mientras que TTHH y NELLY logran un FPR de menos de $2,7\,\%$. Count-less no informa ningún flujo de ratón como un elefante.

5.9.4. Tiempo de Clasificación

En la ingeniería de tráfico es muy importante clasificar rápidamente los flujos elefante para procesarlos y optimizar el enrutamiento. La mayoría de los algoritmos, excepto Count-less, se basan en los primeros N paquetes del flujo para clasificarlo. P4Tree, NELLY y DL-classification2 consiguen un tiempo de clasificación muy bajo. En promedio, este tiempo fue de $0,6s$ para la clasificación en DL-classification2 ($N = 5$) y de $0,8s$ para P4Tree y Nelly ($N = 7$). Los otros trabajos aumentan significativamente el tiempo de clasificación, pHeavy por ejemplo toma 2,6s y aunque usa ML su modelo de clasificación necesita los primeros 20 paquetes para proporcionar una buena precisión. Count-less se basa en contadores, esto le permite establecer un umbral en términos de peso o número de paquetes. Para comparar el algoritmo, asumimos un umbral de elefante $\theta_L = 100k$ y como resultado, el tiempo de clasificación es de 3,8s, que representan más de $4x$ del tiempo de un algoritmo basado en ML como P4Tree. Además, aumentar el umbral de θ_L también dará como resultado un aumento en el tiempo. Finalmente, TTHH ($N = 13$) obtiene un tiempo de clasificación de 8,5s y es el más alto porque este enfoque no considera θ_{TO}.

5.9.5. Modificaciones de Red

Los trabajos en el servidor como NELLY requieren la instalación de software adicional en todos los servidores, lo cual representa problemas de escalabilidad. Los enfoques del plano de datos como P4Tree, Count-less y pHeavy requieren modificaciones a nivel de hardware, ya que se deben implementar *switches* programables en la red [116]. Sin embargo, la renovación de *switches* convencionales a *switches* programables es el camino hacia las redes de próxima generación [117].

5.9.6. Factores de Desempeño

Dependiendo de la ubicación del método de clasificación en la red, varios factores pueden afectar su rendimiento. Los enfoques del lado del controlador, como TTHH, dependen de los recursos disponibles en el controlador (*i.e.*, procesamiento, memoria y ancho de banda) para evitar una sobrecarga de procesamiento, ya que deben ejecutar algoritmos complejos mientras realizan sus funciones predeterminadas. Los enfoques del lado del servidor como NELLY también consumen muchos recursos, por lo que se necesitan servidores potentes para garantizar una buena precisión. P4Tree, pHeavy y Count-less dependen del rendimiento de procesamiento de paquetes del *switch* programable y de la complejidad del programa P4. Además, la memoria del *switch* también es extremadamente importante, ya que limita la cantidad de flujos que se pueden monitorear al mismo tiempo y, por lo tanto, afecta directamente las métricas de precisión.

Nota de autor: *Este capitulo presento la evaluación de P4Tree. Inicialmente, describió el prototipo y los lenguajes de programación y librerías que este emplea. Luego, presento las trazas de trafico real UNIV1 y UNIV2, las cuales son empleadas para el entrenamiento y la evaluación del modelo RF de P4Tree. Posteriormente, presento las métricas de rendimiento y la configuración del experimento. Finalmente, analizo cuantitativamente los resultados de la evaluación y comparo P4Tree con los enfoques de clasificación de flujos elefante y ratón más representativos en la literatura.*

Capítulo 6

Conclusiones y Trabajos Futuros

6.1. Conclusiones

En este trabajo de grado se presenta la respuesta a la pregunta: **¿Cómo clasificar flujos elefante y ratón en Planos de Datos programables considerando el balance entre precisión y tiempo de clasificación?**.

La hipótesis inicial teoriza que utilizar ML en Planos de Datos programables permite una clasificación de flujos elefante y ratón, equilibrando la precisión y el tiempo de clasificación. Para abordar este reto y responder la pregunta investigación, se diseña un mecanismo basado en Random Forest para clasificar flujos elefante y ratón en Planos de Datos programables (*i.e.*, P4Tree). Se implementa el prototipo del mecanismo en el lenguaje de programación de *switches* P4 y finalmente, se evalúa el rendimiento del mecanismo en un ambiente emulado. Los resultados muestran lo siguiente:

- P4Tree hace inferencias precisas con unos pocos paquetes del flujo. Clasifica bien tanto elefantes como ratones y permite alcanzar precisiones por encima del 95 % manipulando la importancia de los elefantes ante el RF.
- En promedio, P4Tree clasifica flujos en 0,8s y hasta donde sabemos es el trabajo en el Plano de Datos que clasifica más rápido los flujos elefante.

- El modelo RF de P4Tree es flexible, puesto que permite operar con un modelo de más o menos árboles dependiendo del desempeño de los *switches* programables y la velocidad del tráfico que circula la red.

- P4Tree es comparable con otros enfoques en la literatura, además, es escalable. Ya que para operar solo necesita el despliegue de algunos *switches* PISA en la red. Sin embargo, para determinar la viabilidad de su implementación en un DCN o una red real, hacen falta más pruebas en una implementación con *switches* programables reales.

En general, el mecanismo de clasificación de flujos propuesto en esta disertación es preciso, rápido y corrobora que ML puede ser usado para clasificar elefantes y ratones equilibrando la precisión y el tiempo de clasificación.

6.2. Trabajos Futuros

- Implementar y evaluar P4Tree en un *switch* programable real como Barefoot Tofino [69] para determinar la viabilidad de su implementación en una red real.

- Explorar e implementar nuevos algoritmos ML y nuevas características del flujo como IAT (*Inter-Arrival Time*) para mejorar la precisión de P4Tree.

- Emplear el método de actualización de modelos de P4Tree para implementar algoritmos incrementales como ARF en el Plano de Datos programable.

- Implementar en el Plano de Datos programable un enfoque complementario para P4Tree, el cual aproveche su rápida clasificación de flujos para tomar decisiones en pro de la red, e.g., un balanceador de carga como [26], un mecanismo de enrutamiento inteligente de flujos elefante como [22] o un mecanismo de enrutamiento inteligente de trafico sensible a la latencia como [24].

Bibliografía

[1] Z. Hang, M. Wen, Y. Shi, and C.-y. Zhang, "Programming protocol-independent packet processors high-level programming (p4hlp): Towards unified high-level programming for a commodity programmable switch," *Electronics*, vol. 8, p. 958, 08 2019.

[2] K. chan Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics," *Computer Networks*, vol. 50, no. 1, pp. 46 – 62, 2006.

[3] M. G. Alcalá Casillas, "La galaxia internet: reflexiones sobre internet, empresa y sociedad, de manuel castells," *Revista mexicana de ciencias políticas y sociales*, vol. 62, no. 231, pp. 407–412, 2017.

[4] J. Marcus, "The economic impact of internet traffic growth on network operators," *SSRN Electronic Journal*, 01 2014.

[5] A. Feldmann, O. Gasser, F. Lichtblau, E. Pujol, I. Poese, C. Dietzel, D. Wagner, M. Wichtlhuber, J. Tapiador, N. Vallina-Rodriguez, O. Hohlfeld, and G. Smaragdakis, "Implications of the COVID-19 Pandemic on the Internet Traffic," in *IEEE (ITG-Symposium)*, 2021, pp. 1–5.

[6] Q. Zhou, Y. He, K. Yang, and T. Chi, "12.3 Exploring PUF-Controlled PA Spectral Regrowth for Physical-Layer Identification of IoT Nodes," in *IEEE International Solid- State Circuits Conference*, vol. 64, 2021, pp. 204–206.

[7] N. M. Karie, N. M. Sahri, and P. Haskell-Dowland, "IoT Threat Detection Advances, Challenges and Future Directions," in *Workshop on Emerging Technologies for Security in IoT*, 2020, pp. 22–29.

[8] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang, and M. F. Zhani, "Data Center Network Virtualization: A Survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 909–928, 2013.

[9] S. P. Khedkar and R. AroulCanessane, "SDN enabled cloud, IoT and DCNs: A comprehensive Survey," in *5th International Conference On Computing, Communication, Control And Automation*, 2019, pp. 1–5.

[10] M. R. Abbasi, A. Guleria, and M. S. Devi, "Traffic engineering in software defined networks: a survey," *Journal of Telecommunications and Information Technology*, no. 4, p. 3–14, 2016.

[11] S. Pan, Y. Zhou, Z. Zhang, S. Yang, F. Qian, and G. Hu, "Identify Congested Links with Network Tomography Under Multipath Routing," *Journal of Network and Systems Management*, vol. 27, no. 2, pp. 409–429, Apr 2019.

[12] Z. Guo, J. Duan, and Y. Yang, "On-Line Multicast Scheduling with Bounded Congestion in Fat-Tree Data Center Networks," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 1, pp. 102–115, 2014.

[13] C. E. Hopps, "Analysis of an equal-cost multi-path algorithm," *RFC Editor*, vol. 2992, pp. 1–8, 2000.

[14] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers," in *Proceedings of the 9th European Conference on Computer Systems*. ACM, 2014.

[15] B. Fortz and M. Thorup, "Internet traffic engineering by optimizing OSPF weights," in *IEEE INFOCOM*, vol. 2, 2000, pp. 519–528 vol.2.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," *Special Interest Group on Data Communication*, vol. 38, no. 4, p. 63–74, 2008.

[17] B. Mao, Z. M. Fadlullah, F. Tang, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "A Tensor Based Deep Learning Technique for Intelligent Packet Routing," in *IEEE Global Communications Conference*, 2017, pp. 1–6.

[18] A. Wahab, N. Ahmad, and J. Schormans, "Variation in QoE of Passive Gaming Video Streaming for Different Packet Loss Ratios," in *Twelfth International Conference on Quality of Multimedia Experience*, 2020, pp. 1–4.

[19] A. A. Barakabitze, A. Ahmad, R. Mijumbi, and A. Hines, "5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges," *Computer Networks*, vol. 167, p. 106984, 2020.

[20] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," in *Proceedings of the Seventh Conference on emerging Networking Experiments and Technologies*, 2011, pp. 1–12.

[21] M. Kiran, B. Mohammed, and N. Krishnaswamy, "DeepRoute: Herding Elephant and Mice Flows with Reinforcement Learning," in *International Conference on Machine Learning for Networking.* Springer, 2019, pp. 296–314.

[22] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, "Online flow size prediction for improved network routing," in *IEEE 24th International Conference on Network Protocols*, 2016, pp. 1–6.

[23] H. Yahyaoui, S. Aidi, and M. F. Zhani, "On using flow classification to optimize traffic routing in sdn networks," in *IEEE 17th Annual Consumer Communications Networking Conference*, 2020, pp. 1–6.

[24] F. Amezquita-Suarez, F. Estrada-Solano, N. L. S. da Fonseca, and O. M. C. Rendon, "An Efficient Mice Flow Routing Algorithm for Data Centers Based on Software-Defined Networking," in *IEEE International Conference on Communications*, 2019, pp. 1–6.

[25] R. Trestian, G. Muntean, and K. Katrinis, "MiceTrap: Scalable traffic engineering of datacenter mice flows using OpenFlow," in *IFIP/IEEE International Symposium on Integrated Network Management*, 2013, pp. 904–907.

[26] J. Liu, J. Li, G. Shou, Y. Hu, Z. Guo, and W. Dai, "Sdn based load balancing mechanism for elephant flow in data center networks," in *International Symposium on Wireless Personal Multimedia Communications*, 2014, pp. 486–490.

[27] R. K. Singh, N. S. Chaudhari, and K. Saxena, "Load balancing in ip/mpls networks: a survey," *Scientific Research Publishing*, 2012.

[28] N. Giri, V. Kukreja, D. Panchi, J. Sajnani, and H. Seedani, "Performance evaluation of load balancing algorithms for sdn," in *4th International Conference On Computing, Communication, Control And Automation*, 2018, pp. 1–4.

[29] X. Ma, L. Liao, Z. Li, and H.-C. Chao, "Asynchronous federated learning for elephant flow detection in software defined networking systems," *Journal of Physics: Conference Series*, vol. 2216, p. 012085, 2022.

[30] F. Tang, H. Zhang, L. T. Yang, and L. Chen, "Elephant flow detection and load-balanced routing with efficient sampling and classification," *IEEE Transactions on Cloud Computing*, vol. 9, no. 3, pp. 1022–1036, 2021.

[31] A. Pekar, A. Duque-Torres, W. K. Seah, and O. M. Caicedo Rendon, "Towards threshold-agnostic heavy-hitter classification," *International Journal of Network Management*, vol. 32, no. 3, p. e2188, 2022.

[32] F. Estrada-Solano, O. M. Caicedo, and N. L. S. Da Fonseca, "Nelly: Flow detection using incremental learning at the server side of sdn-based data centers," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 2, pp. 1362–1372, 2020.

[33] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *IEEE INFOCOM*, 2011, pp. 1629–1637.

[34] B. Turkovic, J. Oostenbrink, and F. Kuipers, "Detecting heavy hitters in the data-plane," *arXiv preprint arXiv:1902.06993*, 2019.

[35] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pheavy: Predicting heavy flows in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.

[36] X. Z. Khooi, L. Csikor, J. Li, M. S. Kang, and D. M. Divakaran, "Revisiting heavy-hitter detection on commodity programmable switches," in *IEEE 7th International Conference on Network Softwarization*, 2021, pp. 79–87.

[37] P. R. Torres, A. García-Martínez, M. Bagnulo, and E. P. Ribeiro, "An elephant in the room: Using sampling for detecting heavy-hitters in programmable switches," *IEEE Access*, vol. 9, pp. 94 122–94 131, 2021.

[38] S. Kim, C. Jung, R. Jang, D. Mohaisen, and D. Nyang, "Count-less: A counting sketch for the data plane of high speed switches," *Computing Research Repository*, vol. abs/2111.02759, 2021.

[39] I. Butun, Y. K. Tuncel, and K. Oztoprak, "Application layer packet processing using pisa switches," *Sensors*, vol. 21, no. 23, p. 8010, 2021.

[40] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM*, vol. 44, no. 3, p. 87–95, 2014.

[41] J. Sonchack, D. Loehr, J. Rexford, and D. Walker, "Lucid: A language for control in the data plane," in *ACM SIGCOMM.* ACM, 2021, p. 731–747.

[42] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Iisy: Practical in-network classification," *arXiv preprint arXiv:2205.08243*, 2022.

[43] M. Hamdan, B. Mohammed, U. Humayun, A. Abdelaziz, S. Khan, M. A. Ali, M. Imran, and M. N. Marsono, "Flow-Aware Elephant Flow Detection for Software-Defined Networks," *IEEE Access*, vol. 8, pp. 72 585–72 597, 2020.

[44] W.-X. Liu, J. Cai, Y. Wang, Q. C. Chen, and J.-Q. Zeng, "Fine-grained flow classification using deep learning for software defined data center networks," *Journal of Network and Computer Applications*, vol. 168, p. 102766, 2020.

[45] K. A. Simpson, R. Cziva, and D. P. Pezaros, “Seiðr: Dataplane Assisted Flow Classification Using ML,” in *IEEE Global Communications Conference*, 2020, pp. 1–6.

[46] B. Dai, G. Xu, B. Huang, P. Qin, and Y. Xu, “Enabling network innovation in data center networks with software defined networking: A survey,” *Journal of Network and Computer Applications*, vol. 94, pp. 33–49, 2017.

[47] M. Al-Fares, A. Loukissas, and A. Vahdat, “A scalable, commodity data center network architecture,” in *ACM SIGCOMM*. ACM, 2008, p. 63–74.

[48] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “Vl2: A scalable and flexible data center network,” *ACM SIGCOMM*, vol. 39, no. 4, p. 51–62, 2009.

[49] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” *ACM SIGCOMM*, vol. 45, no. 4, p. 123–137, 2015.

[50] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network.” ACM, 2015, p. 183–197.

[51] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, “Contra: A programmable system for performance-aware routing,” in *17th USENIX Symposium on Networked Systems Design and Implementation*. Santa Clara, CA: USENIX Association, 2020, pp. 701–721.

[52] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, “Hula: Scalable load balancing using programmable data planes,” in *Proceedings of the Symposium on SDN Research*. ACM, 2016.

[53] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, “Scaling hardware accelerated network monitoring to concurrent and dynamic queries with

*Flow," in *USENIX Annual Technical Conference.* Boston, MA: USENIX Association, 2018, pp. 823–835.

[54] B. J. Van Asten, N. L. van Adrichem, and F. A. Kuipers, "Scalability and resilience of software-defined networking: An overview," *arXiv preprint arXiv:1408.6760*, 2014.

[55] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[56] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf, F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos, "Knowledge-defined networking," *ACM SIGCOMM*, vol. 47, no. 3, p. 2–10, 2017.

[57] D. M. Casas-Velasco, O. M. C. Rendon, and N. L. S. da Fonseca, "Intelligent routing based on reinforcement learning for software-defined networking," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 870–881, 2021.

[58] S. M. Kasongo and Y. Sun, "A deep learning method with wrapper based feature extraction for wireless intrusion detection system," *Computers Security*, vol. 92, p. 101752, 2020.

[59] N. Ranjan, S. Bhandari, H. P. Zhao, H. Kim, and P. Khan, "City-wide traffic congestion prediction based on cnn, lstm and transpose cnn," *IEEE Access*, vol. 8, pp. 81 606–81 620, 2020.

[60] S. Ayoubi, N. Limam, M. A. Salahuddin, N. Shahriar, R. Boutaba, F. Estrada-Solano, and O. M. Caicedo, "Machine learning for cognitive network management," *IEEE Communications Magazine*, vol. 56, no. 1, pp. 158–165, 2018.

[61] J. Hyun and J. W.-K. Hong, "Knowledge-defined networking using in-band network telemetry," in *19th Asia-Pacific Network Operations and Management Symposium*, 2017, pp. 54–57.

[62] S. Yan, A. Aguado, Y. Ou, R. Wang, R. Nejabati, and D. Simeonidou, "Multi-layer network analytics with sdn-based monitoring framework," *J. Opt. Commun. Netw.*, vol. 9, no. 2, pp. A271–A279, 2017.

[63] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," *STD 77, RFC 7011*, 2013.

[64] C. Lin, C. Chen, J. Chang, and Y. H. Chu, "Elephant flow detection in datacenters using OpenFlow-based Hierarchical Statistics Pulling," in *2014 IEEE Global Communications Conference*, 2014, pp. 2264–2269.

[65] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The Nature of Data Center Traffic: Measurements Analysis," in *ACM SIGCOMM.* ACM, 2009, p. 202–208.

[66] T. Benson, A. Akella, and D. A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *ACM SIGCOMM.* ACM, 2010, p. 267–280.

[67] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *IEEE INFOCOM*, 2011, pp. 1629–1637.

[68] X. Li and C. Qian, "Low-complexity multi-resource packet scheduling for network function virtualization," in *IEEE INFOCOM*, 2015, pp. 1400–1408.

[69] Edgecore Networks, "WEDGE 100BF-32X 100GBE DATA CENTER SWITCH," https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335, 2021.

[70] Netberg, "Programmable switches for Academia and Research," https://netbergtw.com/top-support/barefoot-faster-program/, 2021.

[71] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," vol. 44, no. 4, p. 503–514, 2014.

[72] I. Butun, Y. K. Tuncel, and K. Oztoprak, "Application layer packet processing using pisa switches," *Sensors*, vol. 21, no. 23, 2021.

[73] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM.* ACM, 2013, p. 99–110.

[74] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu, "DC.P4: Programming the Forwarding Plane of a Data-Center Switch," in *ACM SIGCOMM.* ACM, 2015.

[75] P4.org, "Let's get started," https://opennetworking.org/p4/lets-get-started/, 2015.

[76] R. Datta, S. Choi, A. Chowdhary, and Y. Park, "P4guard: Designing p4 based firewall," in *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, 2018, pp. 1–6.

[77] P. L. Consortium *et al.*, "P4 runtime: A control plane framework and tools for the p4 programming language," *Website, https://github. com/p4lang/PI*, 2018.

[78] H. Harkous, K. Sherkawi, M. Jarschel, R. Pries, M. He, and W. Kellerer, "P4rcprobe for evaluating the performance of p4runtime-based controllers," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 74–80.

[79] D. Bhat, J. Anderson, P. Ruth, M. Zink, and K. Keahey, "Application-based qoe support with p4 and openflow," in *IEEE INFOCOM WKSHPS*, 2019, pp. 817–823.

[80] S. Kodeswaran, M. T. Arashloo, P. Tammana, and J. Rexford, "Tracking P4 Program Execution in the Data Plane," in *Proceedings of the Symposium on SDN Research.* ACM, 2020, p. 117–122.

[81] R. Teixeira, R. Harrison, A. Gupta, and J. Rexford, “Packetscope: Monitoring the packet lifecycle inside a switch,” in *Proceedings of the Symposium on SDN Research*, 2020, pp. 76–82.

[82] M. A. Ridwan, N. A. M. Radzi, F. Abdullah, and Y. E. Jalil, “Applications of machine learning in networking: A survey of current issues and future challenges,” *IEEE Access*, vol. 9, pp. 52 523–52 556, 2021.

[83] C. Jiang, H. Zhang, Y. Ren, Z. Han, K.-C. Chen, and L. Hanzo, “Machine learning paradigms for next-generation wireless networks,” *IEEE Wireless Communications*, vol. 24, no. 2, pp. 98–105, 2017.

[84] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, “Taurus: A data plane architecture for per-packet ml,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2022, p. 1099–1114.

[85] R. Proietti, X. Chen, K. Zhang, G. Liu, M. Shamsabardeh, A. Castro, L. Velasco, Z. Zhu, and S. J. B. Yoo, “Experimental demonstration of machine-learning-aided qot estimation in multi-domain elastic optical networks with alien wavelengths,” *J. Opt. Commun. Netw.*, vol. 11, no. 1, pp. A1–A10, Jan 2019.

[86] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, “Accelerating distributed reinforcement learning with in-switch computing,” in *ACM/IEEE 46th Annual International Symposium on Computer Architecture*, 2019, pp. 279–291.

[87] D. M. Casas-Velasco, O. M. C. Rendon, and N. L. S. da Fonseca, “Drsir: A deep reinforcement learning approach for routing in software-defined networking,” *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.

[88] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, “pforest: In-network inference with random forests,” *arXiv preprint arXiv:1909.05680*, 2019.

[89] Q. Qin, K. Poularakis, K. K. Leung, and L. Tassiulas, “Line-speed and scalable intrusion detection at the network edge via federated learning,” in *IFIP Networking Conference (Networking)*, 2020, pp. 352–360.

[90] C. Romero, J. L. Olmo, and S. Ventura, “A meta-learning approach for recommending a subset of white-box classification algorithms for moodle datasets,” in *Educational Data Mining 2013*, 2013.

[91] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, “Scikit-learn: Machine learning in python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[92] E. Frank, M. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg, “Weka-a machine learning workbench for data mining,” in *Data mining and knowledge discovery handbook.* Springer, 2009, pp. 1269–1277.

[93] N. Shukla and K. Fricklas, *Machine learning with TensorFlow.* Manning Greenwich, 2018.

[94] P. L. Consortium, “Behavioral model (bmv2),” GitHub, 2022, accessed abr, 2022. [Online]. Available: https://github.com/p4lang/behavioral-model

[95] X. Z. Khooi, L. Csikor, J. Li, M. S. Kang, and D. M. Divakaran, “Revisiting heavy-hitter detection on commodity programmable switches,” in *IEEE 7th International Conference on Network Softwarization*, 2021, pp. 79–87.

[96] A. Cutler, D. R. Cutler, and J. R. Stevens, *Random Forests.* Boston, MA: Springer US, 2012, pp. 157–175.

[97] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Yanhui Geng, Li Chen, K. Chen, and Hao Jin, “Online flow size prediction for improved network routing,” in *IEEE 24th International Conference on Network Protocols*, 2016, pp. 1–6.

[98] Z. Liu, D. Gao, Y. Liu, H. Zhang, and C. H. Foh, “An adaptive approach for elephant flow detection with the rapidly changing traffic in data center network,” *International Journal of Network Management*, vol. 27, no. 6, 2017.

[99] X. Ma, L. X. Liao, Z. Li, and H.-C. Chao, "Asynchronous federated learning for elephant flow detection in software defined networking systems," in *Journal of Physics: Conference Series*, vol. 2216, no. 1. IOP Publishing, 2022, p. 012085.

[100] D. Sanvito, A. Marchini, I. Filippini, and A. Capone, "Cedro: an in-switch elephant flows rescheduling scheme for data-centers," in *6th IEEE Conference on Network Softwarization.*

[101] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proceedings of the Symposium on SDN Research.* ACM, 2017, p. 164–176.

[102] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-Wide Heavy Hitter Detection with Commodity Switches," in *Proceedings of the Symposium on SDN Research.* Proceedings of the Symposium on SDN Research ACM, 2018.

[103] Y. Huang, W. Shih, and J. Huang, "A classification-based elephant flow detection method using application round on sdn environments," in *APNOMS*, 2017, pp. 231–234.

[104] S. Chao, K. C. Lin, and M. Chen, "Flow Classification for Software-Defined Data Centers Using Stream Mining," *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 105–116, 2019.

[105] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *ACM SIGCOMM.* ACM, 2010, p. 267–280.

[106] J.-H. Lee and K. Singh, "Switchtree: In-network computing and traffic analyses with random forests," *Neural Computing and Applications*, pp. 1–12, 11 2020.

[107] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, and G. Carle, "Cryptographic hashing in p4 data planes," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 09 2019, pp. 1–6.

[108] R. R. S, R. R, M. Moharir, and S. G, "Scapy- a powerful interactive packet manipulation program," in *international conference on networking, embedded and wireless systems*, 2018, pp. 1–5.

[109] W. McKinney *et al.*, “Data structures for statistical computing in python,” in *Proceedings of the 9th Python in Science Conference*, vol. 445, no. 1, 2010, pp. 51–56.

[110] D. C. Muñoz and F. A. Saavedra, “P4Tree repository,” GitHub, 2022, accessed Mar, 2022. [Online]. Available: https://github.com/freddysaav/P4Tree

[111] A. Zarek, Y. Ganjali, and D. Lie, “Openflow timeouts demystified,” *Univ. of Toronto, Toronto, Ontario, Canada*, 2012.

[112] A. Turner, M. Bing, and F. Klassen, “Tcpreplay–pcap editing and replaying utilities,” 2013. [Online]. Available: http://tcpreplay.appneta.com

[113] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, “Using mininet for emulation and prototyping software-defined networks,” in *IEEE Colombian conference on communications and computing*, 2014, pp. 1–6.

[114] P. Poupart, Z. Chen, P. Jaini, F. Fung, H. Susanto, Y. Geng, L. Chen, K. Chen, and H. Jin, “Online flow size prediction for improved network routing,” *IEEE 24th International Conference on Network Protocols*, pp. 1–6, 2016.

[115] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *ACM SIGCOMM*. ACM, 2010, p. 63–74.

[116] D. Ding, M. Savi, G. Antichi, and D. Siracusa, “Incremental deployment of programmable switches for network-wide heavy-hitter detection,” in *IEEE 7th International Conference on Network Softwarization*, 2019, pp. 160–168.

[117] G. Bueno, M. Saquetti, P. Rodrigues, I. Lamb, L. Gaspary, M. C. Luizelli, M. F. Zhani, J. R. Azambuja, and W. Cordeiro, “Managing virtual programmable switches: Principles, requirements, and design directions,” *IEEE Communications Magazine*, vol. 60, no. 2, pp. 53–59, 2022.

Clasificación de flujos elefante y ratón en planos de datos programables

ANEXOS

Trabajo de pregrado

David Camilo Muñoz Garcia
Freddy Andres Saavedra Hoyos

Director: PhD. Oscar Mauricio Caicedo Rendón
Codirector: PhD. Felipe Estrada Solano

Departamento de Telemática
Facultad de Ingeniería Electrónica y Telecomunicaciones
Universidad del Cauca
Popayán, Cauca, 2022

Anexo A

Prototipo

El Anexo A presenta el enlace de redireccionamiento al repositorio en GitHub, en el cual están los respectivos códigos fuente de P4Tree.

`https://github.com/freddysaav/P4Tree`

Anexo B

Publicación

El Anexo B presenta el artículo enviado a la comunidad científica con el propósito de que sea publicado.

- **David Camilo Muñoz Garcia**, **Freddy Andres Saavedra Hoyos**, Oscar Mauricio Caicedo Rendón, Felipe Estada Solano. **P4Tree: Approach based on Random Forest for Elephant Flows Classifying in Programmable Data Plane**. Elsevier - Journal of Network and Computer Applications.
    - Estado: Escrito y enviado.
    - Clasificación: A1
    - Factor de Impacto: 7.574
    - Índice H: 115 Scimago

P4Tree: An Approach based on Random Forests for Classifying Elephant Flows in Programmable Data Planes

David Muñoz-Garcia[a], Freddy Saavedra-Hoyos[a], Oscar M. Caicedo[a], Felipe Estrada-Solano[b] and Katherine Casilimas[a]

[a]*Grupo de Ingeniería Telemática, Universidad del Cauca, Popayán 190002, Colombia*
[b]*LOGICIEL, Fundación Universitaria de Popayán, Popayán 190003, Colombia*

ARTICLE INFO

Keywords:
Elephant flow classification
Software defined networking
Programmable data plane
P4
Random forest

Abstract

The significant volume, variety, and speed of traffic generated by the advent of the Internet of Things and the exponential inclusion of new applications and services in the network require support in Data Centers. Conventional routing techniques in Data Centers such as Equal Cost Multi-Path can degrade network performance when handling mouse and elephant flows. This compromises compliance with the latency, Quality of Service, and Quality of Experience requirements of the data center and directly affects the performance of the applications and services it supports. For this reason, novel techniques have been proposed to classify elephant flows and process them for the benefit of the network. Different approaches employ Machine Learning in the Software Defined Networking Control Plane, network servers, or Data Plane to classify flows. However, these methods lead to traffic overhead, low scalability, and high classification time. This paper introduces a novel approach based on Random Forest to quickly and accurately classify elephant flows in the programmable Data Plane called P4Tree. P4Tree trains a Random Forest model offline on the Knowledge Plane and deploys it in the Data Plane to classify flows at line speed. P4Tree RF model can be updated at runtime and is flexible since it allows for operation with a model of more or fewer trees depending on the switches performance and the traffic conditions. P4Tree is extensively evaluated based on real traffic traces and different Random Forest configurations. The evaluation results show that P4Tree is accurate and achieves a low classification time.

1. Introduction

The Internet has become a basic necessity for the population with access to technology. It generates a massive and constant consumption of the products and services hosted there, making it the biggest and most efficient money-making 'machine' today (Alcalá Casillas, 2017). Online consumerism creates the need to innovate, so every day, new applications and services are indexed on the net (Feldmann et al., 2021), significantly increasing the traffic that circulates on it and thus the use of computer and storage resources. Traffic will become even more relevant in terms of variety, speed, and volume with the advent of the Internet of Things (IoT), which by 2025, aims to connect 75 billion devices to the network (Zhou et al., 2021)(Karie et al., 2020) (from light bulbs and smart home appliances to industrial machines and self-driving cars). IoT is supported by the bandwidth provided by Data Center Networks (DCN) (Bari et al., 2013), which are currently planning their infrastructure to meet the massive and imminent demand for connectivity (Khedkar and AroulCanessane, 2019).

*Corresponding author

davidcmunoz@unicauca.edu.co (D. Muñoz-Garcia); freddysaa@unicauca.edu.co (F. Saavedra-Hoyos); omcaicedo@unicauca.edu.co (Oscar M. Caicedo); felipe.estrada@docente.fup.edu.co (F. Estrada-Solano); lkcasilimas@unicauca.edu.co (K. Casilimas)

ORCID(s): 0000-0002-5474-2486 (D. Muñoz-Garcia); 0000-0003-0088-3694 (F. Saavedra-Hoyos); 0000-0003-2223-947X (Oscar M. Caicedo); 0000-0003-4797-9204 (F. Estrada-Solano); 0000-0003-3717-0057 (K. Casilimas)

Avoiding the congestion of growing network traffic is one of the main challenges shared by conventional networks and DCN (Pan et al., 2019). Traditional protocols such as Equal Cost Multi-Path (ECMP) (Hopps, 2000), Weighted Cost Multi-Path (WCMP) (Zhou et al., 2014) and Open Shortest Path First (OSPF) (Fortz and Thorup, 2000), route without regard to links utilization. Moreover, they do not discriminate between elephant flows (large and long-lived) and mouse flows (small and short) (Mao et al., 2017), generating congestion on some links by allocating more elephants on the same route (hot-spots). This situation compromises the performance, latency, Quality of Service (QoS), and Quality of Experience (QoE) of applications (*e.g.*, industrial control, remote control, etc.), services (*e.g.*, streaming, online video games, etc.), and in the future, of applications framed in 5G use cases (*e.g.*, remote surgery, autonomous driving, etc.) (Wahab et al., 2020) (Barakabitze et al., 2020). In this context, the elephant flow classification plays an essential role in traffic engineering (Kiran et al., 2019). Mainly for intelligent routing of elephant flows (Yahyaoui et al., 2020), intelligent routing of latency-sensitive traffic (Amezquita-Suarez et al., 2019) and load balancers (Qin et al., 2021).

The versatility and programmability of Software-Defined Networking (SDN) allow classifying elephant flows at different planes in the network (Giri et al., 2018). The works (Ma et al., 2022)(Tang et al., 2021)(Pekar et al., 2022) classify flows in the Control Plane using techniques based on Machine Learning (ML). These contributions obtain good accuracy in classifying elephants. However, collecting flow information from the switches introduces high

classification time and overhead (*i.e.*, communication overhead between the switches and the controller). A high classification time delays decision-making in mechanisms that depend on elephant flow classification (Amezquita-Suarez et al., 2019)(Yahyaoui et al., 2020)(Qin et al., 2021), degrading their performance. Works like (Estrada-Solano et al., 2020)(Curtis et al., 2011) propose to migrate flow classification to the network servers. These methods reduce the classification time and overhead without sacrificing accuracy. However, it is necessary to modify the operating system of each server, which generates scalability problems. Furthermore, its correct operation depends directly on the server's resources (*i.e.*, processing and storage).

On the programmable Data Plane, the works (Khooi et al., 2021)(Torres et al., 2021)(Kim et al., 2021) propose packet-count based classifiers. These classifiers achieve very high accuracy at the cost of high classification time and high memory consumption due to the long periods that the flow information must be stored inside the switch (*i.e.*, until the flow exceeds a fixed weight or number of packets threshold). The only job (as far as we know) that classifies elephant flows with ML on the programmable Data Plane (Zhang et al., 2021) improves classification time over counter-based classifiers. However, the Decision Tree (DT) model used in this work must collect 20 packets from a flow to accurately classify it. Therefore, a model that requires fewer packets to classify a flow can considerably reduce its classification time. Hybrid approaches like (Hamdan et al., 2021)(Hamdan et al., 2020)(Liu et al., 2020)(Simpson et al., 2020) employ optimized counters in the programmable Data Plane and ML in the controller to corroborate decisions made with specific flows, which increases the accuracy and flexibility of the mechanism for any traffic. However, the inclusion of the controller implies extra classification time and overhead.

In this paper, we propose a novel approach for elephant flows classification, called P4Tree. which applies Random Forest (RF) in the programmable Data Plane to quickly and accurately classify flows. P4Tree trains an RF model offline on the Knowledge Plane (Casas-Velasco et al., 2021). The trained model is sent to the controller, which uses P4Runtime (Consortium et al., 2018) to install it on the programmable Data Plane, specifically, on ToR switches. These switches apply the RF model to classify the elephant flows online and report them to the controller for further processing. P4Tree allows for retraining the model in the Knowledge Plane and installing it on the switches at runtime. Furthermore, its RF model is flexible since it allows to operate with a model of more or fewer trees depending on the classification performance requirements (*i.e.*, accuracy and time), the switches performance (*i.e.*, the packet processing capacity), and traffic speed. We exhaustively evaluated the performance of P4Tree using the real traffic traces UNIV1 and UNIV2 (Benson et al., 2010) and different RF configurations regarding the number of trees. The evaluation results show that P4Tree makes accurate inferences with a few packets and can meet the line speed classification time scale. Therefore, it quickly reports the elephants to the controller so they can be intelligently processed as soon as possible.

The remainder of this document is as follows. Section 2 introduces P4Tree. Section 3 presents a quantitative evaluation of P4Tree using different RF configurations and real packet traces. Section 4 compares P4Tree with related work. Section 5 concludes the paper and presents further works.

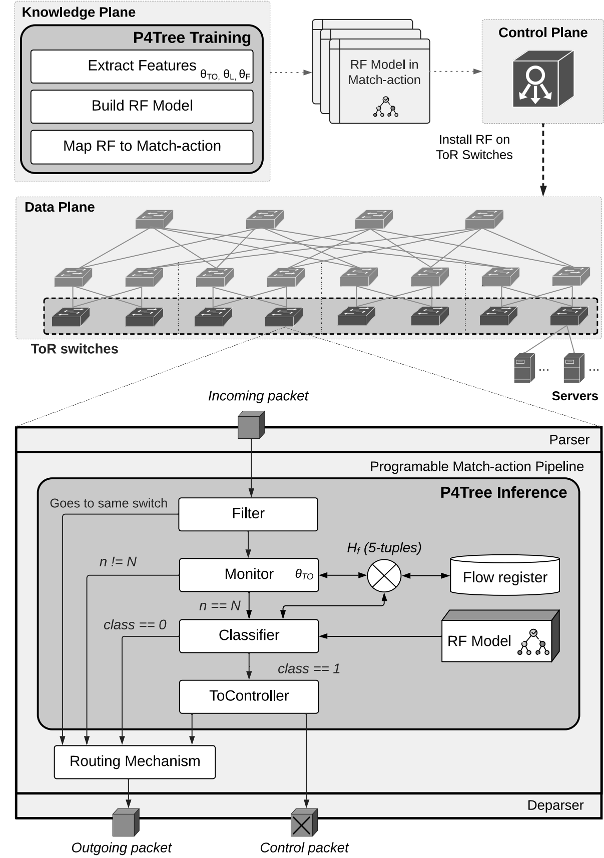

Figure 1: P4Tree Architecture

2. P4Tree

Figure 1 presents P4Tree, a flow classification method that applies RF in the programmable Data Plane to quickly and accurately classify elephant flows. P4Tree operates on Top-of-Rack (ToR) switches of tree-based topologies like Fat-tree (Al-Fares et al., 2008) and can be installed on DCN or conventional networks. P4Tree uses an RF model for classification, as it is an accurate and transparent algorithm, *i.e.*, it is possible to know the decision made by the algorithm and the process to reach it (Romero et al., 2013). Also, RF is built from basic operations and conditionals, allowing it to be encoded on Protocol Independent Switch Architecture (PISA) switches (Butun et al., 2021) using a Data Plane programming language like P4 (Bosshart et al., 2014) or Lucid (Sonchack et al., 2021).

P4Tree is divided into two parts, P4Tree training, and P4Tree inference. First, P4Tree training aims to offline train

the RF model on the Knowledge Plane and then send it to the controller, which installs it inside the ToR switches. Also, if necessary (*i.e.*, if the network traffic characteristics change and the classification threshold must be updated), P4Tree training can retrain the model offline and reinstall it on the switches. Second, P4Tree Inference applies the previously trained model to classify the flows in the programmable Data Plane online. Also, it is responsible for reporting flows classified as elephants to the controller for further processing (*e.g.*, intelligent routing of elephant flows, intelligent routing of latency-sensitive traffic, and load balancers).

2.1. P4Tree Training

To offline train the RF model, P4Tree Training divides its operation into three modules: *Extract Features*, *Build Model* and *Map RF to Match-action*.

Extract Features: This module transforms real traffic traces into training datasets. For this, it receives packet traces in Packet Capture (PCAP) format as input. Which it processes to conform the flows from *5-tuple* header (*i.e.*, source and destination IP, source and destination port, and protocol) and extract the features of each one in a new Comma-Separated Values (CSV) format dataset. The features that define the P4Tree training dataset are protocol, source port, destination port, and the weight of the first N packets of a flow. This module labels the datasets with zeros and ones to differentiate mouse and elephant flows, respectively. For this purpose, a classification threshold θ_L is applied to the total weight of the flow (*i.e.*, if a flow is greater than θ_L, it is an elephant). It should be noted that this module considers a TimeOut (θ_{TO}) to terminate flows that have not received packets for a specific time. Moreover, only flows with a weight greater than θ_F were considered for creating the datasets. This is done to avoid training the model with flows that are definitely latency-sensitive mice and to ensure that each sample has all the features necessary for classification, *i.e.*, to ensure that the weights of the first N packets of the flow have already been collected.

Build Model: This module train an RF model from a dataset created by the Extract Features module. For this, it takes advantage of the Knowledge Plane power computing to use a training environment (Pedregosa et al., 2011) that allows building RF models and configuring the training parameters. Initially, this module trains the model without limiting the number of trees, the trees' maximum depth, or the trees' maximum leaf nodes. Then, it refines these parameters until an optimal value is obtained. That is when the RF obtains good accuracy and a good classification time.

Map RF to Match-action : Like Busse-Grawitz et al. (2019), P4Tree's RF model works from two components: *program code*, which runs on programmable PISA switches, and *program configuration*, which specifies the behavior of this program. The difference between code and configuration is that PISA switches allow changing the configuration at runtime while changing the code requires a switch restart. For this reason, P4Tree only encodes the generalized structure of the RF model and keeps the parameters that define the behavior of a specific RF model reconfigurable. Changing the configuration of compiled programs on PISA switches can be done in two ways: by modifying registers or entries in match-action tables. In this context, this module maps the trained RF model to match-action entries, which configure the RF model of the compiled program in the ToR switches (*i.e.*, P4Tree Inference).

To encode an RF model in a switch, it is necessary to define the structure and semantics of RF in the packet processing model of PISA switches. In these switches, all packet processing logic is defined in a programmable pipeline, which itself is a sequence of match-action tables that allow their entries to be updated from the Control Plane (Bosshart et al., 2014). RF algorithm is an assembly of different DTs, and each tree is a set of decision nodes organized descendingly in levels (Cutler et al., 2012). Each match-action stage of the switch pipeline is used to embed a particular level of a DT. A decision node compares a feature with a threshold specified at each tree level. If the threshold is exceeded, the processing is passed to the right node in the next level (*i.e.*, the next match-action stage). Otherwise, it passes to the left node, where the threshold comparison process is repeated, and so on. Then, at the last level of the tree, the leaf node assigns a label to the sample. As RF consists of a group of n different DTs, n labels are calculated, and the most repeated will be considered as the final decision. Listing 1 presents the structure of a table at the *n*th level of a DT.

Listing 1: Match-action table encoding a node at the *n*th level of a RF tree

```
table level_n {
        key = {
            meta.prev_nodeID: exact;
            meta.prev_feature: exact;
            meta.was_true: exact;
        }
            actions = {
                CheckFeature;
                SetClass;
                NoAction;
        }
        size = 1024;
}
```

The key (*i.e.*, the match) used in tables representing DT nodes is: the unique identifier of the previous node, the identifier of the feature evaluated in the previous node, and the previous result (*i.e.*, if the condition was met or not). The key is based on the previous values to identify the current node's exact position in the tree structure and what specific action this node performs. This key maps to actions such as CheckFeature, SetClass, and NoAction. Decision nodes perform CheckFeature, which compares a feature with a threshold specified. While the leaf nodes perform SetClass, which assigns a label to the sample (*i.e.*, one or zero).

Listing 2: Decision node (line 1) and leaf node (line 2) in the semantics of match-action entries

```
1. Key(prev_nodeID, prev_feature, was_true) =>
                          CheckFeature(feature, threshold)

2. Key(prev_nodeID, prev_feature, was_true) =>
                                          SetClass(class)
```

Note that the exact content of the tables will be populated by this module. For this, it uses the script provided by Lee and Singh (2020) to map each node of the trained RF model in a match-action entry (see Listing 2). Each entry defines the exact values of the key (left side) and the input parameters of its action (right side), *i.e.*, the feature and threshold for decision nodes and the label for leaf nodes. Once this module has mapped the RF Model to match-action entries, it sends them to the controller, which installs them on the ToR switches and enables the RF model of the compiled program (*i.e.*, P4Tree Inference) to classify.

2.2. P4Tree Inference

P4tree inference applies the RF model installed on the ToR switches to online classify elephant flows and report them to the controller. As illustrated in Figure 1, P4Tree inference performs the online classification from four modules located in the match-action programmable pipeline of the switch: *Filter*, *Monitor*, *Classifier*, and *ToController*. In addition, the network Routing Mechanism is also defined in this pipeline. It is responsible for routing all the traffic to its destination and can be any mechanism defined by the network manager, *e.g.*, ECMP, WCMP (Ye et al., 2018), or CEDRO (Sanvito et al., 2020). In Algorithm 1, the process of each P4tree inference module is detailed.

When a packet arrives at the switch, the parser extracts the packet header. The parser is a PISA architecture module that allows the programmer to specify the header fields that the switch recognizes and thus define the protocols supported by the switch (Butun et al., 2021). The parser only extracts IPv4 headers since, as far as we know, there are no publicly available IPv6 traffic traces or datasets that allow evaluating the behavior of P4Tree. Then, the match-action programmable pipeline uses the extracted header to process the packet logically. In this pipeline, P4Tree Inference starts processing the packet with the Filter module.

The *Filter* (Algorithm 1, lines 1-4) avoid additional processing of all those packets that are addressed to a server on the same switch and also avoid storing information about these packets inside the switch. For this, the Filter compares the destination IP of the packet with the IPs of the switch interfaces. On the one hand, if the destination of the packet is directly connected (*i.e.*, it should not jump to another switch to reach its destination), the Routing Mechanism immediately applies the preconfigured routing on the network (*i.e.*, specifies through which switch port the packet should go out) and quickly queues the packet in the buffer to be sent to its destination. This is so that the packet is not processed by the other P4Tree Inference modules, which consume extra processing and storage resources. On the other hand, if the packet's destination is a server on another switch, the *Monitor* module continues to process the packet.

The *Monitor* (Algorithm 1, lines 5-17) manages the *FlowRegister*, which is the database that stores the information of the flows that cross the switch. P4Tree stores each observed flow in a position of the *FlowRegister*, and in each position are stored: the 5-tuple header, the weight of

Algorithm 1: P4Tree Inference

input : incoming packet header h_p, packet size s_p, timestamp t_p and flow size classification model m
output: either packet p and control packet p_c to report elephant flows
data : flow timeout threshold θ_{TO}, number of packages needed to classify N and hash function H_f

```
1  begin process h_p, s_p, t_p
       // Filter
2      int [ ] ← vector of switch interfaces;
3      if h_p.ip_dst ∈ int [ ] then
4          f ← call SimpleForwarding(p);
5      else
           // Monitor
6          get 5-tuple from h_p
7          fidx ← calculate FlowINDEX from 5-tuple and H_f;
8          if FlowRegister[ fidx ] is ∅ then
9              f ← call Init_Flow(fidx, 5-tuple, t_p, s_p)
10         else
11             F ← fetch last flow f ∈ FlowRegister such that f.idx = fidx;
12             if (currentTime − f.lastSeenTime) > θ_TO then
13                 f ← call Init_Flow(fidx, 5-tuple, t_p, s_p)
14             else
15                 f ← call Update_Flow(f, fidx, t_p, s_p)
16             end
17         end
           // Classifier
18         n ← current number of packets in the fidx position;
19         features ← get flow features from FlowRegister;
20         if n == N then
21             f.class ← m.Classify(features);
22             update f → FlowRegister;
23         end
24         class ← current flow class;
25         if class == 1 then
               // ToController
26             p_c ← create a control package from 5-tuple;
27             f ← call ToController(p_c);
28             f ← call SimpleForwarding(p);
29         else
30             f ← call SimpleForwarding(p);
31         end
32     end
33 end
34 function Init_Flow(fidx, 5-tuple, t_p, s_p):
35     f ← initialize a new flow in FlowINDEX position fidx;
36     f.5tuple [ ] ← array of flow header fields from 5-tuple;
37     f.startTime ← f.lastSeenTime ← t_p;
38     f.size ← f.sizePackets[0] ← s_p;
39     create f → FlowRegister;
40     return f
41 end
42 function Update_Flow(f, fidx, s_p, t_p):
43     n ← current number of packets in the fidx position;
44     if n ≤ N then
45         f.sizePackets[n] ← s_p;
46     end
47     f.size ← f.size + s_p;
48     f.lastSeenTime ← t_p;
49     update f → FlowRegister;
50     return f
51 end
```

the first N packets, the last-seen packet time, and the flow class, which by default is mouse. In line 6, the *Monitor* starts to track the flows from each incoming packet's 5-tuples header, size, and timestamp. In line 7, the *Monitor* applies a Hash function over the 5-tuples header to generate the FlowINDEX position where the flow will be stored. The programmable Data Plane provides different Hash algorithms that can be used to transform the 5-tuples into a new fixed-length value, *e.g.*, crc32, crc16, cksum16, and xor16 (Scholz et al., 2019). P4Tree uses the crc32 algorithm since this Hash obtains better performance given its greater number of bits. In lines 8 to 18 the *Monitor* checks if the FlowINDEX position is empty in the *FlowRegister*. If so, the Monitor initiates a new flow (Lines 34-40). Otherwise, two things can happen if there is already a flow at this position. First, the elapsed time since the arrival of the last packet of the flow is greater than θ_{TO}. In this case, the Monitor overwrites the new flow in the FlowINDEX position (Line 13) to terminate inactive flows and optimize memory usage. Second, the elapsed time since the arrival of the last packet in the flow is less than θ_{TO}. In this case, the *Monitor* updates the flow's last-seen packet time and the weight of the first N packets (lines 42-50).

P4Tree avoids classifying a large number of mouse flows (generally latency sensitive) since the next module (*i.e.*, the *Classifier*) is only applied when the N packet of the flow arrives. Before its arrival, it is not necessary to classify the flow; after, it is only necessary to store the flow so as not to reclassify it. Therefore, for all packets other than N, the *Monitor* directly passes the packet to the Routing Mechanism to route it as soon as possible.

The *Classifier* module (Algorithm 1, Lines 18-31) classify the flows based on the RF model previously installed by P4Tree Training in the switch. When the N packet of a flow arrives, the *Classifier* extracts from the *FlowRegister* the necessary features for the classification (*i.e.*, protocol, source port, destination port, and the weight of the first *N* packets of a flow). Then, it labels the flow from the installed RF model (see Subsection 2.1). And finally, update the calculated class in *FlowRegister*. If the *Classifier* labels a flow as mouse, the Routing Mechanism applies the preconfigured routing, and the packet goes out through the corresponding port. Otherwise, if the flow is an elephant, the next module (*i.e.*, *ToController*) reports it to the controller. It should be noted that the P4Tree RF model allows reconfiguring the match-action entries in runtime so that the RF model can be updated if necessary, that is, if the traffic characteristics in the network change and the threshold θ_L must be updated. For this, P4Tree Training must rebuild the model offline and install it back on the switches.

The *ToController* module (Algorithm 1, Lines 25-29) report flows labeled as elephant to the controller. To report these flows, it creates a control packet that includes the 5-tuple header of the flow and then routes it to the controller. Where other network approaches can take advantage of this information to make decisions, *e.g.*, intelligent elephant flow routing mechanisms can handle elephants to avoid link congestion (Yahyaoui et al., 2020), load balancers can precisely balance links (Qin et al., 2021), or latency-sensitive traffic routing mechanisms can create routing rules that favor mice (Amezquita-Suarez et al., 2019).

3. Evaluation

This section presents the P4Tree evaluation. Section 3.1 shows its prototype, while sections 3.2 and 3.3 present the datasets and performance metrics, respectively. Section 3.4 shows the experiment setup and sections 3.5, 3.6, and 3.7 discuss the results of the evaluation. Finally, section 3.8 presents the remarks of the quantitative evaluation.

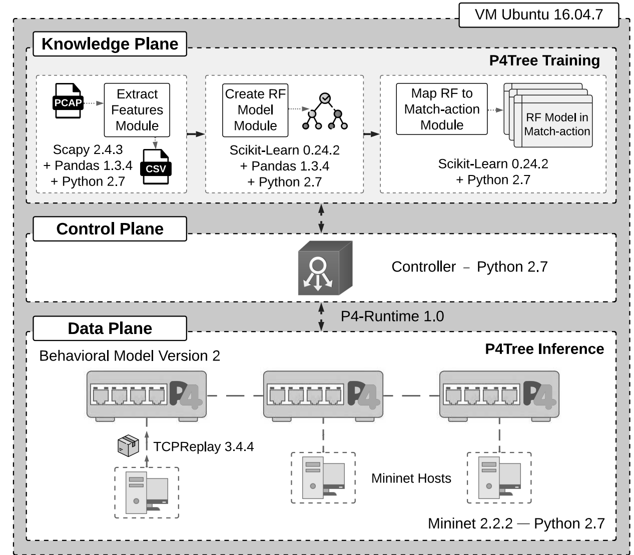

Figure 2: P4Tree prototype

3.1. Prototype

Figure 2 presents the P4Tree prototype. The Knowledge Plane where P4Tree Training operates offline is fully developed in Python 2.7 with the Scapy 2.4.3 (S et al., 2018), Pandas 1.3.4 (McKinney et al., 2010) and Scikit-Learn 0.24.2 (Pedregosa et al., 2011) libraries. The Scapy library is a powerful interactive network packet manipulation tool. Pandas is an easy-to-use library specialized in manipulating and analyzing data structures. Scikit-Learn is an open-source ML library that provides simple and efficient tools for data preprocessing, model training, model evaluation, and model export. The Data Plane where P4Tree inference operates online is programmed in $P4_{16}$ (Bosshart et al., 2014). This language allows to write a program capable of specifying how PISA switches process packets, so it brings all the benefits of software engineering to the edge of the network (*i.e.*, program writing, debugging, and code coverage). Furthermore, by handling metadata, registers, conditionals, counters, and match-action tables, it is possible to create complex programs for PISA switches as ML algorithms (Zheng et al., 2022). In the Control Plane, we

Table 1
DETAILS OF PACKET TRACES AND IPV4 FLOWS OBTAINED USING THE 5-TUPLE HEADER AND $\theta_{TO} = 5$S

Packet traces		UNIV1		UNIV2	
Duration		3914.62s		9480.35s	
Data size		12 GB		75 GB	
Packets		19,855,388		100,000,000	
Data byte rate		3303 kBps		7984 kBps	
Data bit rate		26 Mbps		63 Mbps	
IPv4 % of total traffic		98.98% *(mostly TCP)*		99.9% *(mostly UDP)*	
IPv4 flows		1.02 M		1.04 M	
Details of IPv4 flows	Flow size	% of IPv4 flows	% of IPv4 traffic	% of IPv4 flows	% of IPv4 traffic
	≥ 10 kB	7.16%	95.06%	5.91%	98.81%
	≥ 100 kB	0.83%	83.71%	1.93%	96.86%

developed a Python controller, which has been implemented from the examples in (Consortium, 2022). The controller communicates with the Data Plane via the P4Runtime API (Consortium et al., 2018). This API provides a standard way to control the Data Plane elements of a device or program defined by a P4 program, *e.g.*, to add and delete entries to a switch's match-action tables. The P4Tree prototype is available at (Muñoz-Garcia and Saavedra-Hoyos, 2022a).

3.2. Datasets

For the training and evaluation of P4Tree, the real traffic traces UNIV1 and UNIV2 (Benson et al., 2010) were used. Both traces have been captured in university data centers and belong to different services, *e.g.*, system backups, hosting distributed file systems, e-mail servers, and Web services. 99% of the packets in the traces correspond to IPv4 traffic and include mainly TCP and UDP flows. Table 1 summarizes the characteristics of UNIV1 and UNIV2.

P4Tree Training uses the original traces in PCAP format to create the RF model training datasets in CSV format. To create a dataset, P4Tree only needs to define the parameters: θ_{TO}, N, and θ_L. P4Tree uses the 5-tuple header and θ_{TO} to form the flows. $\theta_{TO} = 5s$ was established based on the OpenFlow Timeouts analysis considered by Zarek et al. (2012). Like Estrada-Solano et al. (2020), P4Tree only classify flows greater than $\theta_F = 10kB$ for two reasons. First, below this value are more than 92% of flows (*i.e.*, flows that are definitely latency-sensitive mice). Second, above this value are all potential elephants (*i.e.*, flows carrying 95% of total traffic). As such, for a flow to weigh more than $10kB$, it must have a minimum of N = 7 packets. Flows with less than 7 packets by default are considered mouse to avoid unnecessary processing. Therefore, they are removed from the training datasets. The classification threshold $\theta_L = 100kB$ was set to label flows as mice or elephants and represent the ground truth. This value was chosen to compare the performance of P4Tree with other works since a standard threshold has not been defined in the literature, and it continues to be a research challenge (Pekar et al., 2022). Training datasets are available in (Muñoz-Garcia and Saavedra-Hoyos, 2022b).

The original traces in PCAP format are also used to evaluate the performance of P4Tree classifying traffic at line speed. For this, the traces are introduced into the network using Tcpreplay (Turner et al., 2013) and classified in real-time by P4Tree Inference. Tcpreplay is a set of utilities for editing and replaying pre-captured network traffic in PCAP format. In addition, it allows to increase or decrease the original speed of the traffic traces, set a maximum speed value, and even control the number of packets it reproduces in a second.

3.3. Performance metrics

The P4Tree performance is evaluated using the metrics: True Positive Rate (TPR), False Positive Rate (FPR), the Matthews Correlation Coefficient (MCC), the Labeling Time (T_L), and Losses. Labeling time and the accuracy metrics TPR, FPR, and MCC are used in the literature to evaluate elephant classifiers (Estrada-Solano et al., 2020). While Losses are used to evaluate classifiers in the Data Plane (Lee and Singh, 2020).

Considering that flows classified as elephants are reported to the controller for further processing, P4Tree aims to identify as many elephants as possible, affecting as few mice as possible (usually latency-sensitive). Therefore, Elephant flows are considered as the positive condition. From this, TPR specifies the percentage of well-classified elephants, while FPR describes the percentage of misclassified mice. TPR and FPR are metrics derived from the confusion matrix, providing a measure between 0 and 1 (Chicco et al., 2021).

The MCC is used to analyze the balance between TPR and FPR. For this purpose, it provides a measure between -1 and 1 from the confusion matrix, which only approaches 1 when the algorithm classifies both elephants and mice well (*i.e.*, when maximizing the TPR and FPR is minimized). If the MCC is below 0, the algorithm is less accurate than a random classifier (Chicco and Jurman, 2020).

The classification time T_C represents the period from when the first packet of a flow arrives until P4Tree labels it as elephant or mouse (Estrada-Solano et al., 2020) and is described by the equation $T_C = T_{fc} + T_L$. Where T_{fc} specifies the flow features collection time and depends directly on the time it takes for the first 7 packets to arrive at the switch. On the other hand, T_L represents the time it takes for the P4Tree RF model to process the features and label a flow. T_L is used for P4Tree performance analysis, while T_C was used for P4Tree comparative analysis.

Table 2
PERFORMANCE OF P4TREE WITH DIFFERENT NUMBERS OF TREES TO CLASSIFY FLOWS AS MICE AND ELEPHANTS

Algorithm	UNIV1 - $W_1 = 6$				UNIV2 - $W_1 = 2$			
	TPR(%)	FPR(%)	MCC	$T_L(\mu s)$	TPR(%)	FPR(%)	MCC	$T_L(\mu s)$
1-tree RF	87.1	4	0.36	9.7	75.1	5.3	0.39	14.3
3-trees RF	88.2	4	0.36	31.2	75.2	5.1	0.4	42.7
5-trees RF	88.7	3.5	0.38	54.8	75.3	5	0.4	68.1
7-trees RF	89.5	3.3	0.39	71.4	76.2	4.9	0.41	87.1
9-trees RF	90.1	3.1	0.4	90.6	78.4	4.5	0.43	103.7

Losses represent the percentage of packets the switch drops when its buffer queue is full (Lee and Singh, 2020). This percentage depends on the speed of the traffic and the ability of P4Tree to process individual packets. Losses are calculated by comparing the number of packets that enter the switch with the number of successfully processed packets that leave the switch to their destination.

3.4. Experiment setup

We deployed the Control, Knowledge, and Data Planes over an Ubuntu 16.04.7 machine, with a Core i7-9700 processor, 8 CPU cores, and 16 GB of RAM. A simple network with a programmable switch and two hosts are emulated just like Lee and Singh (2020) and Kim et al. (2021). For this, Mininet 2.2.2 (de Oliveira et al., 2014) is used with the P4 software switch called Behavior Model Version 2 (bmv2).

Bmv2 (Consortium, 2022) is a powerful tool for developing, testing, and debugging P4 Data Planes and Control Plane software written for them. However, the performance of bmv2 in terms of throughput and latency is affected by two factors. First, how fast bmv2 can process individual packets: the Packet Processing Time (PPT) in bmv2 depends on the performance of the machine's hardware (*i.e.*, how many CPU cores, how much CPU cache, etc.) and the complexity of the P4 program that compiles the switch (*i.e.*, the number of code lines). If the PPT is high, the buffer queue flows slower, and the switch drops packets' risk increases. Better performance should be expected on a real switch (Consortium, 2022). Second, the speed of traffic crossing the switch: if the traffic speed is high enough, the buffer queue fills up and forces the switch to drop packets.

To evaluate the performance of P4tree, the RF model is trained on the Knowledge Plane with the default Scikit-Learn settings for RF, except for the following parameters. The maximum depth of each tree and the maximum number of leaf nodes were set to 15 and 200, respectively, to avoid large and slow trees. The number of trees varies in the experiments. The weight W_0 associated with the negative class (*i.e.*, the mouse class) is kept by default (*i.e.*, 1), and the weight W_1 is associated with the positive class (*i.e.*, the elephant class) varies across experiments. It should be noted that W_0 and W_1 represent the importance the model gives to each class (Poupart et al., 2016). Therefore, the classifier learns more about the class with a higher weight.

3.5. Loss Analysis

A stress test was performed to evaluate the performance of P4Tree processing packets at line speed with respect to the number of trees of the RF model. Which consists of sending the original traffic traces (*i.e.*, UNIV1 and UNIV2) from one host to another through the switch and observing the Losses. Then, the original speed of the traces is increased and decreased to find the maximum speed at which P4Tree presents an optimal performance. Figure 3 shows Losses when P4Tree uses an RF model with 1, 3, 5, 7, and 9 trees to classify traffic at different speeds. The results show that Losses increase as the traffic speed increases. In addition, the Losses also increase as the number of trees increases. High-speed traffic quickly fills the buffer queue and forces the switch to drop packets. On the other hand, an RF with more trees requires a larger number of match-action tables to classify a flow. This represents a higher processing cost and has a negative impact on the buffer queue flow, forcing the switch to drop packets. In UNIV1, whose average speed is $26Mbps$, the Losses go up to 6% as the number of trees increases, while the Losses in UNIV2 ($63Mbps$) go up to 50%. For this reason and to evaluate the accuracy metrics (*i.e.*, TPR, FPR, and MCC) under ideal conditions (*i.e.*, in a scenario where P4Tree can correctly process traffic traces without Losses), the traffic traces speed should be reduced by up to 10x using Tcpreplay to ensure near-zero Losses.

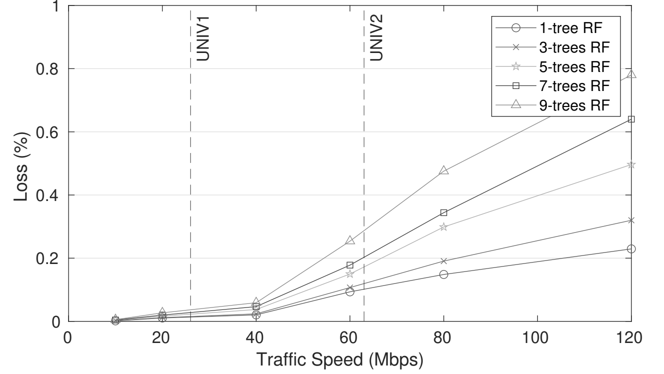

Figure 3: Stress test. The original speed of UNIV1 and UNIV2 is highlighted and it should be noted that P4Tree here runs on a machine and better performance should be expected on a real switch

3.6. Accuracy Analysis

Table 2 presents the TPR, FPR, and MCC when P4Tree uses an RF model with 1, 3, 5, 7, and 9 trees to classify the UNIV1 and UNIV2 traces. The confusion matrix values needed to compute the accuracy metrics are extracted from the registers of programmable switches by P4Runtime. Table 2 only includes results that maximize MCC (*i.e.*, the most balanced configuration) and achieve a minimum TPR of

75%. In UNIV1, the MCC is maximum when the RF model is trained with $W_1 = 6$, while in UNIV2, this is achieved with $W_1 = 2$. The results show that as the number of trees increases, the TPR, FPR, and MCC improve significantly. In UNIV1, P4Tree with a 1-tree RF model obtains a TPR of 87%, while a 9-trees RF model obtains 90%. In UNIV2, the TPR varies from 75% to 78%. The FPR also improves when the RF model is more robust. In UNIV1, the FPR varies from 4% to 3% as the number of trees increases, while in UNIV2, it varies from 5.3% to 4.5%. The best results obtained by P4Tree in terms of MCC are 0.4 and 0.43 for UNIV1 and UNIV2, respectively. This is due to the great class imbalance in both traffic traces, *i.e.*, the percentage of samples from mouse flows (more than 98%) is much higher than the percentage from elephant flows (less than 2%). Therefore, the RF model focuses on learning more about the features of mice, which directly affects the trade-off between TPR and FPR (*i.e.*, MCC).

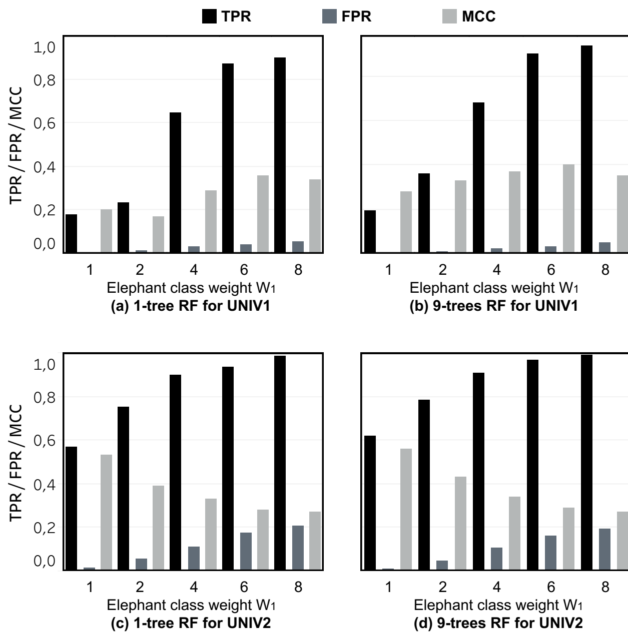

Figure 4: Accuracy of P4Tree with 1-tree RF model (left) and 9-trees RF model (right) when varying the range for the inverse weights of elephant flows (W_1) for the UNIV1 (top) and UNIV2 (bottom)

The 1-tree RF and 9-trees RF models were trained using different weights W_1 to analyze their effect on the accuracy of P4Tree. In Figure 4, W_1 varies between 1 and 8, since above this value, the TPR of the P4Tree does not improve, and the FPR increases significantly. The results show that P4Tree achieves a higher TPR as W_1 increases (up to 94% in UNIV1 and 99% in UNIV2). This is because setting higher weights for the elephant class allows the RF model to increase the importance of misclassifying elephants and decrease the importance of misclassifying mice. This results in a more accurate classifier in elephant flows at the expense of a slight FPR degradation. Furthermore, the trade-off between TPR and FPR (*i.e.*, MCC) improves for UNIV1 as W_1 increases, while for UNIV2, it decreases. This is because the class imbalance in UNIV1 is more significant than in UNIV2 (*i.e.*, UNIV1 has fewer samples of the elephant class than UNIV2); in UNIV1, the percentage of elephants represents 0.8% of the entire trace, while in UNIV2, it represents 1.9% (see Table 1).

3.7. Labeling Time Analysis

Table 2 presents the T_L when P4Tree uses an RF model with 1, 3, 5, 7, and 9 trees to classify the UNIV1 and UNIV2 traces. In order to get the T_L from the switch, it is necessary to make modifications to the bmv2 source code to create time counters in the switch since they currently do not have this function. Regarding the T_L in UNIV1, when P4Tree uses a 1-tree RF model, it takes $9.7\mu s$ to label the flow as elephant or mouse. This labeling time is approximately $0.7x$ the PPT on the emulated switch (*i.e.*, $14\mu s$) and represents less than 3.9% of the Round-Trip Time (RTT) in DCN, *i.e.*, $250\mu s$ (Alizadeh et al., 2010). As shown in Table 2, the T_L increases as the number of trees increases, *e.g.*, a 3-trees RF model requires $31.2\mu s$, while a 5-trees RF model takes $54.8\mu s$. This is because each tree uses a fixed amount of match-action tables to generate a label, and as the classification requires more tables, the T_L increases. In UNIV2, when P4Tree uses a 1-tree RF model, it obtains a T_L of $14.3\mu s$, which is $1x$ the PPT and represents 5.7% of the RTT. While a more robust classifier like the 7-trees RF model or the 9-trees RF model requires $87.1\mu s$ and $103.7\mu s$, respectively.

3.8. Remarks

To conclude regarding the configuration that achieves the best performance in P4Tree, the following aspects must be taken into account:

- There is a trade-off between labeling flows accurately and making it fast. On the one hand, if the RTT of the network is strict, the best option is to use a 3-trees RF model since T_L is below 12.5% of the RTT, and the accuracy metrics are good, *e.g.*, in UNIV1, TPR and FPR represent 88.2% and 4% respectively. On the other hand, if the RTT is flexible, it is possible to use a 5-trees RF model to improve the accuracy metrics significantly, *e.g.*, in UNIV1, both TPR and FPR improve 0.5%.
- P4Tree supports flexible weight configuration to meet different TPR and FPR requirements. It is recommended to use the weights that achieve the most balanced configuration (*i.e.*, $W_1 = 6$ for UNIV1 and $W_1 = 2$ for UNIV2). However, if the priority of the network is to classify as many elephants as possible without significantly affecting mice, $W_1 = 8$ can be used for UNIV1 and $W_1 = 4$ for UNIV2, *e.g.*, in UNIV1, TPR improves 3% and FPR degrades 1.3%.
- Losses are directly related to the traffic speed and the number of trees in the P4Tree RF model. However, the influence of the number of trees in the Losses

becomes important after 5 trees, *e.g.*, in UNIV2, 3-trees RF model Loss 2% more packets than 1-tree RF model, while 5-trees RF model Loss 6% more. For this reason, it is recommended to use the 3-trees RF model. Furthermore, better performance is expected with a real switch that can process more packets per second, *e.g.*, the Barefoot Tofino switch (Edgecore Networks, 2021).

4. Comparative Analysis

P4Tree was compared with NELLY (Estrada-Solano et al., 2020), TTHH (Pekar et al., 2022), Count-Less (Kim et al., 2021), DL-classification2 (Liu et al., 2020) and pHeavy (Zhang et al., 2021). NELLY applies incremental learning in the network servers to effectively classify elephant flows. TTHH uses a novel technique based on Packet Size Distribution and Template Matching (TM) to classify the flows on the Control Plane. On the programmable Data Plane, Count-Less uses packet counters to perfectly classify all flows, while pHeavy is the first work to our knowledge that explores ML to classify elephants in the Data Plane. And finally, DL-classification2 is a hybrid approach with pre-classification in the Data Plane and exact classification in the Control Plane. The results from the UNIV1 dataset with 3-trees RF model were used to compare its performance in relation to: classification technique, reported elephants, false elephants, classification time, network modifications, and performance factors.

Classification technique. Elephant flows can be classified from different techniques depending on the plane of the network where the classifier is located. Both on the Control Plane and the servers, it is possible to implement complex algorithms that require all kinds of operations (*e.g.*, multiplications, divisions, floating number operations, etc.) and loops to achieve high accuracy. For this reason, NELLY can implement incremental algorithms such as Adaptive Hoeffding Option Tree and Adaptive Random Forest, DL-classification2 can deploy Convolutional Neural Networks, and TTHH can execute its TM-based technique.

On the other hand, the programmable Data Plane is limited in terms of operations and loops, preventing the above techniques' implementation. All jobs that classify elephants and mice in the Data Plane are counter-based, except for pHeavy and P4Tree (as far as we know). Counters such as Count-Less do not disregard any flow. Therefore they must store the flow information (*i.e.*, the 5-tuple header and packet count) until a set threshold is exceeded. This generates perfect accuracy at the expense of high classification time and excessive memory consumption. P4Tree and pHeavy seek to mitigate these problems by implementing an ML model that precisely classifies the flow with the first N packages.

PHeavy implements a DT in the programmable Data Plane manually mapping the tree with conditionals. This makes it a fixed model that cannot be updated. In addition, its source code is not available to date. P4Tree introduces a method for assembling multiple trees and forming a Random Forest algorithm. This classification model is much more robust, achieves better accuracy metrics and reduces elephant flow classification time. P4Tree also provides a mechanism that allows updating the model from the controller via P4Runtime in order to adapt to changing network traffic.

Reported elephants. It is essential for flow classification methods to report as many elephants as possible. NELLY and TTHH classify and report well over 95% of elephant flows, DL-classification2 exceeds 93%, and pHeavy and P4Tree report 90% and 88%, respectively. However, P4Tree can report more than 92% of elephant flows when W_1 is increased at the cost of slightly degrading the FPR. Count-Less provides perfect classification and reports all elephants since it is based on packet counters.

False elephants. Mice that are erroneously classified as elephant flows are reported to the Control Plane and processed unnecessarily. DL-classification2 misclassifies 5.6% of mice, P4Tree and pHeavy misclassify less than 3.1%, while TTHH and NELLY misclassify less than 2.7%. Count-Less does not report any false elephant.

Classification time. In traffic engineering, it is imperative to classify elephant flows to process them and optimize routing quickly. All algorithms except Count-Less rely on the first N packets of the flow to classify it. P4Tree, NELLY, and DL-classification2 achieve a very low classification time. On average, this time was 0.6s for DL-classification2 ($N = 5$) and $0.8s$ for P4Tree and Nelly ($N = 7$). PHeavy takes $2.6s$, as its classification DT model needs the 20 first packets of a flow to provide good accuracy. Count-Less is based on counters, which allows setting a threshold in terms of weight or number of packets. To compare the algorithm, we assume an elephant threshold $\theta_L = 100k$, and as a result, the classification time is $3.8s$, representing more than $4x$ the time of an ML-based algorithm such as P4Tree. Furthermore, increasing the θ_L threshold will result in a slower classifier. Finally, TTHH ($N = 13$) obtains a classification time of $8.5s$ and is the highest because this approach does not consider the θ_{TO}.

Network modifications. Approaches in the servers like NELLY require installing additional software on all servers, representing scalability problems. Data Plane approaches such as P4Tree, count-less, and pHeavy require hardware-level modifications; programmable PISA switches must be deployed on the network (Ding et al., 2019). Remarkably, the renewal of conventional switches to programmable switches is the way to next generation networks (Bueno et al., 2022).

Performance factors. Depending on the location of the classification method on the network, several factors may affect its performance. Control Plane approaches such as TTHH depend on the resources available in the controller to avoid a processing overload, as they must execute complex algorithms while performing their default functions. Approaches that classify on servers like NELLY consume many resources, so powerful servers are needed to ensure optimal performance (Pekar et al., 2022). P4Tree, pHeavy and count-less depend on the packet processing performance of the programmable switch and the complexity of the P4

program. In addition, the switch's memory is also crucial since it limits the number of flows the switch can monitor simultaneously. Therefore, if the switch lacks memory, it cannot track all the flows and directly affects the accuracy metrics.

5. Conclusion and Future Work

This paper has introduced P4Tree, an approach based on RF to quickly and accurately classify elephant flows in the programmable Data Plane. P4Tree operates on ToR switches and allows to update its RF model in runtime. The P4Tree RF is flexible since it allows to operate with a model of more or fewer trees depending on the switches performance and the traffic conditions. The evaluation demonstrates that P4Tree makes accurate inferences with a few packets and can meet the line speed classification time scale. Therefore, it quickly reports the elephants to the controller so they can be intelligently processed as soon as possible. For future work, we plan to evaluate the performance of P4Tree in a real programmable switch and use the inter-arrival time (IAT) of the packets as features to more accurately classify the flows. Also, we would like to take advantage of our model update method to implement incremental algorithms such as adaptive Random Forest in the programmable Data Plane.

References

Al-Fares, M., Loukissas, A., Vahdat, A., 2008. A scalable, commodity data center network architecture, in: ACM SIGCOMM, pp. 63–74. doi:10.1145/1402958.1402967.

Alcalá Casillas, M.G., 2017. La galaxia internet: reflexiones sobre internet, empresa y sociedad, de manuel castells. Revista mexicana de ciencias políticas y sociales 62, 407–412.

Alizadeh, M., Greenberg, A., Maltz, D.A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., Sridharan, M., 2010. Data center tcp (dctcp), in: ACM SIGCOMM, pp. 63–74. doi:10.1145/1851182.1851192.

Amezquita-Suarez, F., Estrada-Solano, F., da Fonseca, N.L.S., Rendon, O.M.C., 2019. An Efficient Mice Flow Routing Algorithm for Data Centers Based on Software-Defined Networking, in: IEEE International Conference on Communications, pp. 1–6. doi:10.1109/ICC.2019.8761552.

Barakabitze, A.A., Ahmad, A., Mijumbi, R., Hines, A., 2020. 5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges. Computer Networks 167, 106984. doi:10.1016/j.comnet.2019.106984.

Bari, M.F., Boutaba, R., Esteves, R., Granville, L.Z., Podlesny, M., Rabbani, M.G., Zhang, Q., Zhani, M.F., 2013. Data Center Network Virtualization: A Survey. IEEE Communications Surveys Tutorials 15, 909–928. doi:10.1109/SURV.2012.090512.00043.

Benson, T., Akella, A., Maltz, D.A., 2010. Network traffic characteristics of data centers in the wild, in: ACM SIGCOMM, pp. 267–280. doi:10.1145/3503222.3507726.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D., 2014. P4: Programming Protocol-Independent Packet Processors. ACM SIGCOMM 44, 87–95. doi:10.1145/2656877.2656890.

Bueno, G., Saquetti, M., Rodrigues, P., Lamb, I., Gaspary, L., Luizelli, M.C., Zhani, M.F., Azambuja, J.R., Cordeiro, W., 2022. Managing virtual programmable switches: Principles, requirements, and design directions. IEEE Communications Magazine 60, 53–59. doi:10.1109/MCOM.001.2100363.

Busse-Grawitz, C., Meier, R., Dietmüller, A., Bühler, T., Vanbever, L., 2019. pforest: In-network inference with random forests. Computing Research Repository doi:doi.org/10.48550/arXiv.1909.05680.

Butun, I., Tuncel, Y.K., Oztoprak, K., 2021. Application layer packet processing using pisa switches. Sensors 21, 8010. doi:10.3390/s21238010.

Casas-Velasco, D.M., Rendon, O.M.C., da Fonseca, N.L.S., 2021. Intelligent routing based on reinforcement learning for software-defined networking. IEEE Transactions on Network and Service Management 18, 870–881. doi:10.1109/TNSM.2020.3036911.

Chicco, D., Jurman, G., 2020. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. BMC genomics 21, 1–13.

Chicco, D., Tötsch, N., Jurman, G., 2021. The matthews correlation coefficient (mcc) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. BioData mining 14, 1–22. doi:10.1186/s13040-021-00244-z.

Consortium, P.L., 2022. Behavioral model (bmv2). URL: https://github.com/p4lang/behavioral-model. accessed apr, 2022.

Consortium, P.L., et al., 2018. P4 runtime: A control plane framework and tools for the p4 programming language. URL: https://github.com/p4lang/PI. accessed jul, 2022.

Curtis, A.R., Kim, W., Yalagandula, P., 2011. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection, in: IEEE INFOCOM, pp. 1629–1637. doi:10.1109/INFCOM.2011.5934956.

Cutler, A., Cutler, D.R., Stevens, J.R., 2012. Random Forests. Springer US, Boston, MA. pp. 157–175. doi:10.1007/978-1-4419-9326-7_5.

Ding, D., Savi, M., Antichi, G., Siracusa, D., 2019. Incremental deployment of programmable switches for network-wide heavy-hitter detection, in: IEEE Conference on Network Softwarization, pp. 160–168. doi:10.1109/NETSOFT.2019.8806649.

Edgecore Networks, 2021. WEDGE 100BF-32X 100GBE DATA CENTER SWITCH. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.

Estrada-Solano, F., Caicedo, O.M., Da Fonseca, N.L.S., 2020. Nelly: Flow detection using incremental learning at the server side of sdn-based data centers. IEEE Transactions on Industrial Informatics 16, 1362–1372. doi:10.1109/TII.2019.2947291.

Feldmann, A., Gasser, O., Lichtblau, F., Pujol, E., Poese, I., Dietzel, C., Wagner, D., Wichtlhuber, M., Tapiador, J., Vallina-Rodriguez, N., Hohlfeld, O., Smaragdakis, G., 2021. Implications of the COVID-19 Pandemic on the Internet Traffic, in: Broadband Coverage in Germany; 15th ITG-Symposium, pp. 1–5. doi:10.2139/ssrn.2531782.

Fortz, B., Thorup, M., 2000. Internet traffic engineering by optimizing OSPF weights, in: IEEE INFOCOM, pp. 519–528. doi:10.1109/INFCOM.2000.832225.

Giri, N., Kukreja, V., Panchi, D., Sajnani, J., Seedani, H., 2018. Performance evaluation of load balancing algorithms for sdn, in: 4th International Conference on Computing Communication Control and Automation, pp. 1–4. doi:10.1109/ICCUBEA.2018.8697762.

Hamdan, M., Khan, S., Abdelaziz, A., Sadiah, S., Shaikh-Husin, N., Al Otaibi, S., Maple, C., Marsono, M., 2021. Dplbant: Improved load balancing technique based on detection and rerouting of elephant flows in software-defined networks. Computer Communications 180, 315–327. doi:https://doi.org/10.1016/j.comcom.2021.10.013.

Hamdan, M., Mohammed, B., Humayun, U., Abdelaziz, A., Khan, S., Ali, M.A., Imran, M., Marsono, M.N., 2020. Flow-Aware Elephant Flow Detection for Software-Defined Networks. IEEE Access 8, 72585–72597. doi:10.1109/ACCESS.2020.2987977.

Hopps, C.E., 2000. Analysis of an equal-cost multi-path algorithm. RFC Editor 2992, 1–8. doi:10.17487/RFC2992.

Karie, N.M., Sahri, N.M., Haskell-Dowland, P., 2020. IoT Threat Detection Advances, Challenges and Future Directions, in: Workshop on Emerging Technologies for Security in IoT, pp. 22–29. doi:10.1109/ETSecIoT50046.2020.00009.

Khedkar, S.P., AroulCanessane, R., 2019. SDN enabled cloud, IoT and DCNs: A comprehensive Survey, in: 5th International Conference On Computing, Communication, Control And Automation, pp. 1–5. doi:10.1109/ICCUBEA47591.2019.9129091.

Khooi, X.Z., Csikor, L., Li, J., Kang, M.S., Divakaran, D.M., 2021. Revisiting heavy-hitter detection on commodity programmable switches, in: IEEE 7th International Conference on Network Softwarization, pp. 79–87. doi:10.1109/NetSoft51509.2021.9492531.

Kim, S., Jung, C., Jang, R., Mohaisen, D., Nyang, D., 2021. Count-less: A counting sketch for the data plane of high speed switches. Computing Research Repository abs/2111.02759. doi:10.48550/arXiv.2111.02759.

Kiran, M., Mohammed, B., Krishnaswamy, N., 2019. DeepRoute: Herding Elephant and Mice Flows with Reinforcement Learning, in: International Conference on Machine Learning for Networking, pp. 296–314. doi:10.1145/2079296.2079304.

Lee, J.H., Singh, K., 2020. Switchtree: In-network computing and traffic analyses with random forests. Neural Computing and Applications , 1–12doi:10.1007/s00521-020-05440-2.

Liu, W.X., Cai, J., Wang, Y., Chen, Q.C., Zeng, J.Q., 2020. Fine-grained flow classification using deep learning for software defined data center networks. Journal of Network and Computer Applications 168, 1084–8045. doi:10.1016/j.jnca.2020.102766.

Ma, X., Liao, L.X., Li, Z., Chao, H.C., 2022. Asynchronous federated learning for elephant flow detection in software defined networking systems, in: Journal of Physics: Conference Series, p. 012085. doi:10.1088/1742-6596/2216/1/012085.

Mao, B., Fadlullah, Z.M., Tang, F., Kato, N., Akashi, O., Inoue, T., Mizutani, K., 2017. A Tensor Based Deep Learning Technique for Intelligent Packet Routing, in: IEEE Global Communications Conference, pp. 1–6. doi:10.1109/GLOCOM.2017.8254036.

McKinney, W., et al., 2010. Data structures for statistical computing in python, in: Proceedings of the 9th Python in Science Conference, pp. 51–56. doi:10.25080/Majora-92bf1922-00a.

Muñoz-Garcia, Saavedra-Hoyos, 2022a. P4Tree repository. URL: https://github.com/freddysaav/P4Tree. accessed Mar, 2022.

Muñoz-Garcia, Saavedra-Hoyos, 2022b. P4Tree training datasets. URL: https://github.com/freddysaav/P4Tree/datasets. accessed Mar, 2022.

de Oliveira, R.L.S., Schweitzer, C.M., Shinoda, A.A., Prete, L.R., 2014. Using mininet for emulation and prototyping software-defined networks, in: IEEE Colombian conference on communications and computing, pp. 1–6. doi:10.1109/ColComCon.2014.6860404.

Pan, S., Zhou, Y., Zhang, Z., Yang, S., Qian, F., Hu, G., 2019. Identify Congested Links with Network Tomography Under Multipath Routing. Journal of Network and Systems Management 27, 409–429. doi:10.1007/s10922-018-9471-2.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al., 2011. Scikit-learn: Machine learning in python. the Journal of machine Learning research 12, 2825–2830.

Pekar, A., Duque-Torres, A., Seah, W.K., Caicedo Rendon, O.M., 2022. Towards threshold-agnostic heavy-hitter classification. International Journal of Network Management 32, 2188. doi:10.1002/nem.2188.

Poupart, P., Chen, Z., Jaini, P., Fung, F., Susanto, H., Geng, Y., Chen, L., Chen, K., Jin, H., 2016. Online flow size prediction for improved network routing. IEEE 24th International Conference on Network Protocols , 1–6doi:10.1109/ICNP.2016.7785324.

Qin, L., Wei, W., Diao, X., 2021. Flowdecider: End-host driven proactive load balancing for data center networks, in: IEEE 21st International Conference on Communication Technology, pp. 931–936. doi:10.1109/ICCT52962.2021.9658099.

Romero, C., Olmo, J.L., Ventura, S., 2013. A meta-learning approach for recommending a subset of white-box classification algorithms for moodle datasets, in: Educational Data Mining 2013.

S, R.R., R, R., Moharir, M., G, S., 2018. Scapy- a powerful interactive packet manipulation program, in: international conference on networking, embedded and wireless systems, pp. 1–5. doi:10.1109/ICNEWS.2018.8903954.

Sanvito, D., Marchini, A., Filippini, I., Capone, A., 2020. Cedro: an in-switch elephant flows rescheduling scheme for data-centers, in: 6th IEEE Conference on Network Softwarization, pp. 368–376. doi:10.1109/NetSoft48620.2020.9165522.

Scholz, D., Oeldemann, A., Geyer, F., Gallenmüller, S., Stubbe, H., Wild, T., Herkersdorf, A., Carle, G., 2019. Cryptographic hashing in p4 data planes, in: ACM/IEEES ymposium on Architectures for Networking and Communications Systems, pp. 1–6. doi:10.1109/ANCS.2019.8901886.

Simpson, K.A., Cziva, R., Pezaros, D.P., 2020. Seiðr: Dataplane Assisted Flow Classification Using ML, in: IEEE Global Communications Conference, pp. 1–6. doi:10.1109/GLOBECOM42002.2020.9348063.

Sonchack, J., Loehr, D., Rexford, J., Walker, D., 2021. Lucid: A language for control in the data plane, in: ACM SIGCOMM, pp. 731–747. doi:10.1145/3452296.3472903.

Tang, F., Zhang, H., Yang, L.T., Chen, L., 2021. Elephant flow detection and load-balanced routing with efficient sampling and classification. IEEE Transactions on Cloud Computing 9, 1022–1036. doi:10.1109/TCC.2019.2901669.

Torres, P.R., García-Martínez, A., Bagnulo, M., Ribeiro, E.P., 2021. An elephant in the room: Using sampling for detecting heavy-hitters in programmable switches. IEEE Access 9, 94122–94131. doi:10.1109/ACCESS.2021.3092281.

Turner, A., Bing, M., Klassen, F., 2013. Tcpreplay–pcap editing and replaying utilities. URL: http://tcpreplay.appneta.com.

Wahab, A., Ahmad, N., Schormans, J., 2020. Variation in QoE of Passive Gaming Video Streaming for Different Packet Loss Ratios, in: Twelfth International Conference on Quality of Multimedia Experience, pp. 1–4. doi:10.1109/QoMEX48832.2020.9123071.

Yahyaoui, H., Aidi, S., Zhani, M.F., 2020. On using flow classification to optimize traffic routing in sdn networks, in: IEEE 17th Annual Consumer Communications Networking Conference, pp. 1–6. doi:10.1109/CCNC46108.2020.9045216.

Ye, J.L., Chen, C., Huang Chu, Y., 2018. A weighted ecmp load balancing scheme for data centers using p4 switches, in: IEEE 7th International Conference on Cloud Networking, pp. 1–4. doi:10.1109/CloudNet.2018.8549549.

Zarek, A., Ganjali, Y., Lie, D., 2012. Openflow timeouts demystified. Univ. of Toronto, Toronto, Ontario, Canada .

Zhang, X., Cui, L., Tso, F.P., Jia, W., 2021. pheavy: Predicting heavy flows in the programmable data plane. IEEE Transactions on Network and Service Management 18, 4353–4364. doi:10.1109/TNSM.2021.3094514.

Zheng, C., Xiong, Z., Bui, T.T., Kaupmees, S., Bensoussane, R., Bernabeu, A., Vargaftik, S., Ben-Itzhak, Y., Zilberman, N., 2022. Iisy: Practical in-network classification. arXiv preprint arXiv:2205.08243 doi:10.48550/arXiv.2205.08243.

Zhou, J., Tewari, M., Zhu, M., Kabbani, A., Poutievski, L., Singh, A., Vahdat, A., 2014. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers, in: Proceedings of the 9th European Conference on Computer Systems, pp. 1–14. doi:10.1145/2592798.2592803.

Zhou, Q., He, Y., Yang, K., Chi, T., 2021. 12.3 exploring puf-controlled pa spectral regrowth for physical-layer identification of iot nodes, in: IEEE International Solid- State Circuits Conference, pp. 204–206. doi:10.1109/ISSCC42613.2021.9365941.

David Muñoz-Garcia is an engineer in electronics and telecommunications from the Universidad del Cauca (2022), Popayan, Colombia. His research interests include networks and services management, Data Plane programming, 5G networks, reinforcement learning, neural networks, and data engineering.

Freddy Saavedra-Hoyos is an engineer in electronics and telecommunications from the Universidad del Cauca (2022), Popayán, Colombia. His topics of interest include network and service management, data science, data engineerin, and machine learning.

Oscar M. Caicedo is a full professor at the Universidad del Cauca, Colombia, where he is a member of the Telematics Engineering Group. He received his Ph.D. degree in computer science (2015) from the Universidade Federal do Rio Grande do Sul, Brazil, and his M.Sc. in telematics engineering (2006) and his degree in electronics and telecommunications

engineering (2001) from the Universidad del Cauca. His recent research interests include network and service management, network functions virtualization, software-defined networking, machine learning for networking, and network softwarization. He serves as Series Editor for the IEEE Communications Magazine Series on Network Softwarization and Management and Associate Editor of the IEEE Networking Letters. He served and keeps serving as Technical Program Co-Chair for IEEE-sponsored international workshops and conferences.

Felipe Estrada-Solano received the bachelor's degree in electronics and telecommunications engineering and the M.Sc. degree in telematics engineering from the Universidad del Cauca, Colombia, in 2010 and 2016, respectively, and the Ph.D. degrees in telematics engineering and in computer science from the Universidad del Cauca and from the Universidade Estadual de Campinas, Brazil, respectively, in 2022. He is currently an SDN Developer with the R&D Department at iVedha, Canada, and a Research Instructor with the Systems Engineering Program at the Fundación Universitaria de Popayán, Colombia. His topics of interest include network and service management, cloud computing, network virtualization, network softwarization, and machine learning.

Katherine Casilimas received the bachelor's degree in electronics and telecommunications engineering and the M.Sc. degree in telematics engineering from the Universidad del Cauca, Colombia, in 2019 and 2021, respectively. She is currently a Software Quality Tester at Alltic. Her topics of interest include machine learning for testing and networking.