UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Ph.D. Thesis

Internet Traffic Classification for High-Performance and Off-The-Shelf Systems

Author:
Pedro María Santiago del Río

Supervisor:
Prof. Javier Aracil Rico

Madrid, 2013

DOCTORAL THESIS:	Internet Traffic Classification for High-Performance and Off-The-Shelf Systems
AUTHOR:	Pedro María Santiago del Río
SUPERVISOR:	Prof. Javier Aracil Rico

The committee for the defense of this doctoral thesis is composed by:

PRESIDENT:	Dr. Eduardo Magaña Lizarrondo
MEMBERS:	Dr. Francesco Gringoli
	Dr. Luigi Iannone
	Dr. Mikel Izal Azcárate
SECRETARY:	Dr. Jorge E. López de Vergara Méndez

A mis padres y a mi hermano.
A ti, pecosa.

Summary

Network traffic monitoring is of paramount importance for network operators due to the ever-increasing links' speed and users' bandwidth demand. Thus, it has awakened the interest of the research community in the last years. Particularly, traffic classification (i.e., to associate traffic with the application that has generated it) is one of the most relevant monitoring tasks, which provides crucial information to network managers. The heterogeneity and complexity of current networks along with the high link's speeds (typically ranging from 1 Gb/s to 40 Gb/s) make traffic monitoring more difficult. This implies a significant investment on infrastructure, especially on the large-scale networks that require multiple points of measurements, given that traffic monitoring tasks are very demanding in terms of computational power. Undoubtedly, traffic classification has to be enough accurate to achieve its expected usefulness for network management, even when traffic is obfuscated, encrypted and uses arbitrary port numbers. Furthermore, because of the constant evolution of networks, the proposed monitoring tools must be definitively flexible, scalable and able to support higher throughput.

This study aims at analyzing the feasibility of a network traffic monitoring system, and particularly, a traffic classification engine, which fulfills the abovementioned challenges, namely: (i) high-performance, (ii) limited cost, (iii) accuracy, and (iv) scalability. Off-The-Shelf (OTS) systems, based on open-source software and commodity hardware, are presented as a great alternative to specialized hardware, which has been traditionally used for such tasks. Particularly, contemporary Non Uniform Memory Access (NUMA) systems with multi-core architectures as well as modern Network Interface Card (NIC)s with multi-queue capabilities, are shown with potential capac-

ity to cope with accurate traffic classification for high-speed and limited-cost systems. Thus, in this thesis, we thoroughly analyze each module of a typical traffic classification engine (and, more generally, of a network traffic monitoring system), namely: packet sniffing, timestamping, flow handling and classification.

First, we reviewed, evaluated and compared the different proposals for packet sniffing (the first task of any monitoring system), highlighting their similarities and differences as well as their pros and cons. Second, we analyzed other low-level task of paramount importance in network traffic monitoring (especially, in real-time services such as multimedia traffic): packet timestamping. We quantify the inaccuracy of packet timestamping by using novel packet capture engines. We propose two approaches to overcome or mitigate such accuracy limitations. Our proposal achieves the best results respect the rest of solutions, even in several orders of magnitude.

Then, we proposed a statistical classification engine based on software-only and commodity-hardware solutions. The proposal is able to on-line classify at more than 14 Million packets per second (Mpps) and 2.8 Million flows per second (Mfps) in a worst-case scenario and up to 20 Gb/s when monitoring a real backbone link. Such astonishing results are possible thanks to the use of an improved network driver, the use of lightweight statistical classification technique; and an exhaustive tuning of critical parameters of the hardware and the software application. Furthermore, we carefully analyzed the flow handling module (which is a common module in the most of monitoring system) and different Machine Learning (ML) tools as traffic classifiers.

Due to its relevance and popularity, we focus on multimedia traffic classification and monitoring. Specifically, we propose two Voice over IP (VoIP) monitoring systems, one for Session Initiation Protocol (SIP)-based VoIP technology and the other one for Skype traffic, able to process and classify traffic at line-rate in a 10 Gb/s link. Finally, the impact of packet sampling (used to reduce the computational load) on traffic classification is analyzed.

Resumen

La monitorización de tráfico de red es de vital importancia para las operadoras debido a la creciente velocidad de los enlaces y demanda de ancho de banda de los usuarios. Así, ha despertado el interés de la comunidad científica en los últimos años. En particular, la clasificación de tráfico (es decir, asociar el tráfico con la aplicación que lo generó) es una de las tareas más relevante de monitorización que da a los gestores de red gran información. La heterogeneidad y complejidad de las redes actuales junto con las altas velocidades de los enlaces (típicamente, en el rango de 1 Gb/s a 40 Gb/s) hacen cada vez más difícil su monitorización. Esto implica una significativa inversión en infraestructura, especialmente en grandes redes que requieren múltiples puntos de medida, dado que las tareas de monitorización son muy demandantes en términos de potencia computacional. Sin duda, la clasificación de tráfico tiene que ser suficientemente precisa para alcanzar su utilidad esperada para la gestión de tráfico, incluso cuando el tráfico está ofuscado, cifrado y usa puertos arbitrarios. Además, por la constante evolución de las redes, las herramientas de monitorización propuestas tienen que ser claramente flexibles, escalables y capaces de soportar tasas superiores.

El principal objetivo de este estudio es analizar la viabilidad de un sistema de monitorización de tráfico de red, y, en particular, un motor de clasificación de tráfico, que cumpla los mencionados retos, esto es: (i) altas prestaciones, (ii) coste limitado, (iii) precisión y (iv) escalabilidad. Los sistemas de propósito general, basados en software de código abierto y hardware de uso extendido, se presentan como una gran alternativa al hardware especializado, que ha sido usado tradicionalmente para estas tareas. En particular, los actuales sistemas con acceso a memoria no uniforme y arquitecturas multi-core

además de las modernas tarjetas de red con múltiples colas, muestran capacidad potencial para enfrentarse con la clasificación de tráfico precisa en sistemas de altas prestaciones y coste limitado. Así, en esta tesis, se analiza detalladamente cada módulo de un motor de clasificación típico (y, más generalmente, de un sistema de monitorización de tráfico de red), a saber: captura de paquetes, marcado de tiempo, formación de flujos y clasificación.

Primero, se han revisado, evaluado y comparado las diferentes propuestas para la captura de paquetes (la primera tarea de cualquier sistema de monitorización), destacando sus semejanzas y diferencias así como sus pros y sus contras. En segundo lugar, se ha analizado otra tarea de bajo nivel de vital importancia en la monitorización (especialmente, en los servicios de tiempo real como el tráfico multimedia): el marcado de tiempo. Se ha cuantificado la imprecisión del marcado temporal cuando se usan los nuevos motores de captura. Proponemos dos soluciones para superar o mitigar tales limitaciones en la precisión. Nuestra propuesta alcanza los mejores resultados respecto al resto de soluciones, incluso en varios órdenes de magnitud.

Más adelante, proponemos un motor de clasificación estadística basado en soluciones con sólo software y hardware de uso extendido. La propuesta es capaz de clasificar en tiempo real a más de 14 millones de paquetes por segundo y 2,8 millones de flujos por segundo en un escenario de caso peor y hasta a 20 Gb/s monitorizando un enlace troncal real. Tales extraordinarios resultados son posibles gracias al uso de un driver de red mejorado, al uso de una técnica de clasificación estadística ligera y a un exhaustivo ajuste de los parámetros críticos del hardware y de la aplicación software. Además, se ha analizado cuidadosamente el módulo de formación de flujos (el cual es común en la mayoría de sistemas de monitorización) y diferentes técnicas de aprendizaje automático usadas como clasificadores de tráfico.

Debido a su relevancia y popularidad, nos centramos en la clasificación y monitorización del tráfico multimedia. Específicamente, proponemos dos sistemas de monitorización de Voz sobre IP, uno para SIP y otro para Skype, capaces de procesar y clasificar tráfico a tasa de línea en un enlace de 10 Gb/s. Finalmente, se ha analizado el impacto del muestreo de paquetes (usado para reducir la carga computacional) sobre la clasificación de tráfico.

Acknowledgments

First of all, I would like to thank my supervisor, Javier Aracil, for his close collaboration, good advice and confidence with me. The fruitfulness of this work is mainly your duty. Thank you for allowing me to start my researching career here.

Likewise, a special acknowledgment is dedicated to my colleagues from the Lab C113 (and former B209): Javier Ramos, Víctor Moreno, José Luis García, David Muelas; and those that are no longer here: Felipe Mata, José Alberto Hernández, Víctor Lopez, Jaime Garnica, David Madrigal, Bas Huiszoon, Alfredo Salvador, Diego Sánchez, Jaime Fullaondo, Walter Fuertes and Pedro Gómez. Guys, don't worry, the second place in HPCN Comunio League is not so bad.

I would like to also thank my other colleagues from the High Performance Computing and Networking Research Group: Jorge López de Vergara, Paco Gómez, Sergio López, Iván González, Luis de Pedro, Gustavo Sutter, Germán Retamosa, Marco Forconesi, Mario Poyato, Rafael Leira and, for sure, `irenerodriguez.ii.uam.es`.

All my work would not have been possible without the support of the Universidad Autónoma de Madrid and the Departamento de Tecnología Electrónica y de las Comunicaciones (former Departamento de Informática) of the Escuela Politécnica Superior. In addition, I would also like to express my gratitude to the Spanish Ministry of Education for funding this Ph.D. under the F.P.U. fellowship program. I hope that such grant program and other public fundings and investments will be continued in the next years, to provide new scientists and people to achieve a better world.

I would like to show my gratefulness to Dario Rossi for hosting me in

Paris. My internship in Paris was very productive. This stay helped me to grow up, both as a researcher and as a person. Moreover, I learned a lot with the discussions with Dario, and working side by side with him and the rest of colleagues, who in addition helped me a lot to integrate in LINCS. *Grazie!* This absolutely includes all the staff and other students, specially Claudio Testa, Konstantinos Katsaros, Rim Kaddah, Rosa Vilardi, Giuseppe Rossini, Chiara Chirichella and Rafaelle Chiocchetti. Although not in person, I also had the chance to work with Francesco Gringoli during my stay in Paris. Our achievements would not have been possible without his supporting.

Now, let me switch to Spanish in the following, for writing personal acknowledgments:

- En primer lugar, quiero agradecer a mis padres, José Luis y Maribel, por su apoyo y ayuda, por la educación en valores que me han proporcionado. Que sepáis que gracias a vosotros he podido llegar hasta aquí.

- Quisiera agradecer a Toñi, por su paciencia y su comprensión, por estar ahí en los malos y en los buenos momentos. Sin tu apoyo incondicional (y tu sonrisa), esto hubiera sido mucho más difícil. Pecosa, tenemos que celebrarlo! ;).

- Un especial agradecimiento a mi hermano José Luis, por aguantarme durante tantos años, por sus consejos y por su apoyo. Sin ti, no sería la persona ni el científico que soy.

- Quisiera aprovechar estas líneas, para recordar a mis abuelas, María Pérez y María Fernández, y a mi tía Mercedes, que ya no están entre nosotros.

- A mi amigo, desde tiempos pretéritos, Santi. Aunque la distancia hace que no nos veamos tanto como quisiéramos, sé que he podido contar contigo siempre que me hiciera falta.

- A mis amigos de la carrera, Sergio, Ignacio y Jesús. Sé que habéis estado ahí (más cerca o más lejos).

- Quiero dar las gracias a mis amigos del Instituto Ouróboros por sus intensas y edificantes charlas y cafés.

- A mis compañeros de grada de Cordobamanía y de la Sección Tifo. En especial, a Manolo: si no me hubieras arreglado el cargador, no podría haber escrito estas líneas ;).

- A familiares y amig@s que he olvidado mencionar. Seguro que sabréis perdonarme. Una parte de este trabajo también es vuestra.

A todos vosotros, gracias de corazón! To all of you, sincerely thanks!

Contents

List of Figures

List of Tables

Acronyms

ADSL Asymmetric Digital Subscriber Line. 128, 129

API Application Programming Interface. 38, 40, 43, 45, 46, 51, 59, 62, 84, 110

BLINC BLINd Classification. 27, 28

BPF Berkeley Packet Filter. 43

CAIDA The Cooperative Association for Internet Data Analysis. 72, 78, 89, 92, 96, 111, 113, 114

CAPEX Capital Expenditures. 2, 3

CDF Cumulative Distribution Function. 163

CPU Central Processing Unit. 10, 12, 14, 17, 20–22, 32, 33, 39, 43, 44, 50–53, 65, 70, 80, 83, 84, 86–88, 90, 93, 96, 100, 104, 106, 113, 115, 134, 148, 151, 152, 157, 174, 180, 192

CR Coral Reef. 92, 93, 95

CRC Cyclic Redundancy Check. 54, 125

CSIC Consejo Superior de Investigaciones Científicas. 190

DDR3 Double Data Rate type 3. 52, 70, 84, 110, 141, 151, 158

DMA Direct Memory Access. 15, 33, 38, 43, 47, 105, 132

DNA Direct NIC Access. 42, 43, 54–56, 60

DPI Deep Packet Inspection. xvi, 3, 22, 24, 26, 27, 108, 115, 126, 136, 154, 193

DSL Digital Subscriber Line. 128

DSLAM Digital Subscriber Line Access Multiplexer. 96

FPGA Field-Programmable Gate Array. 2, 14, 26, 52, 73, 78

FTP File Transfer Protocol. 25, 126

FUAM Fundación de la Universidad Autónoma de Madrid. 179, 191

GbE Gigabit Ethernet. 3, 10, 11, 32, 52, 70, 71, 79, 84, 90, 103, 110, 125

GPL General Public License. 48

GPRS General Packet Radio Service. 119

GPU Graphic Processing Unit. 13, 14, 21–23, 26, 115, 180, 181, 193

HPCAP High-performance Packet CAPture. 49, 51, 54, 55, 57, 60–62, 70, 103, 106

HPTRAC High-Performance TRAffic Classifier. 77, 115

HW Hardware. 5, 12, 17, 18, 32, 35

I/O Input/Output. 6, 9, 15, 16, 45, 55, 63, 73, 74, 132, 171–173, 180

IOH I/O Hub. 13

IP Internet Protocol. 11, 14, 24, 37, 43, 80, 85, 92, 110, 121, 122, 125, 126, 128, 134, 135, 137, 138, 154, 177, 189

IPFIX IP Flow Information Export. 108, 109

IPSec Internet Protocol Security. 21

IRQ Interrupt ReQuest. 53

ISP Internet Service Provider. 20, 53, 96

KISS Chi-Square Signatures. 27

KPT Kernel-level Polling Thread. 7, 69, 70, 72–74

LAN Local Area Network. 128, 129

LSB Least Significant Bits. 11

MAC Media Access Control. 107

Mfps Million flows per second. viii, 3, 78, 85, 87, 88, 100, 101, 103

ML Machine Learning. viii, 3, 75, 77, 126, 127, 158, 169, 175

Mpps Million packets per second. viii, 2, 3, 20, 32, 42, 54, 55, 59, 77–79, 85–88, 90–96, 99–101, 103, 104, 111, 112, 116, 125, 142, 175, 187

MRTG Multi Router Traffic Grapher. 108, 109

MTU Maximum Transmission Unit. 44

NAPI New API. 16, 17, 31, 32, 38, 43, 47

NAT Network Address Translation. 128

NDA Non-Disclosure Agreement. 96

NIC Network Interface Card. vii, 4, 7, 10, 11, 13, 15–17, 20, 22, 30, 32, 34, 36, 38, 39, 43, 45, 46, 49, 51–53, 55, 69, 70, 74, 79, 80, 83, 84, 103, 105, 106, 110, 115, 132, 141, 142, 147, 156, 169, 173, 174, 179–181

NIDS Network Intrusion Detection System. 19, 22, 23, 36, 76, 109

NTP Network Time Protocol. 66

NUMA Non Uniform Memory Access. vii, 12, 13, 18, 33, 39, 42–44, 51, 53, 134, 151, 152, 157, 158, 180, 193

OPEX Operational Expenditures. 2, 3

OS Operating System. 140

OTS Off-The-Shelf. vii, 2, 4–7, 9, 17, 19, 23, 24, 28–31, 40, 64, 75–78, 97, 102, 106, 115, 117, 147–149, 169, 171–173, 176, 177, 179–181

P2P Peer-to-Peer. 3, 25, 155, 162

PCAP Packet Capture. 71, 107, 109, 110, 156, 158

PCI Peripheral Component Interconnect. 9, 13

PCIe PCI-Express. 13, 39, 51, 52, 80, 110, 151

PESQ Perceptual Evaluation of Speech Quality. 128

PoP Point of Presence. 146

POSIX Portable Operating System Interface. 39

PS PacketShader. 44, 45, 52, 84

PSTN Public Switch Telephony Network. 3, 117, 118, 122, 167

PTP Precision Time Protocol. 11, 12, 66, 180, 192

QoE Quality of Experience. 3, 118, 119, 125, 128, 139, 167

QoS Quality of Service. 3, 5, 24, 76, 118, 119, 127, 130, 132, 167

RAID Redundant Array of Independent Disks. 140, 141, 143, 147

RB Red Black. 94–96, 116, 175, 187

RSS Receive Side Scaling. 10, 11, 17, 32, 35, 37, 46, 52, 54, 61, 70, 79, 80, 82–84, 86–88, 91, 132, 179, 180, 192

RTP Real-time Transport Protocol. 25, 26, 36, 37, 122, 124, 125, 131, 132, 134, 135, 137, 140, 141, 143, 144, 147, 176, 177, 188, 189

RX Receiving. 15–17, 38

SBC Session Border Controller. 122, 124

SDP Session Description Protocol. 25, 122, 125, 134

SDRAM Synchronous Dynamic Random Access Memory. 52, 70, 84, 110, 141

SFP Small Form-factor Pluggable. 52, 71

SIP Session Initiation Protocol. viii, 25, 26, 36, 37, 120–122, 124–126, 131, 132, 134–139, 144, 176, 177, 188, 189

SMP Symmetric MultiProcessor. 12

SMTP Simple Mail Transfer Protocol. 25, 126

SPAN Switched Port Analyzer. 124, 142, 168

SVM Support Vector Machine. 78, 81, 97, 98, 100, 101, 127, 175

SW Software. 4, 12, 17, 23

TCP Transmission Control Protocol. 11, 25, 53, 84, 85, 107, 110, 125, 126, 137, 138, 142, 151, 157, 177, 189

TIWS Telefonica International Wholesale Services. 179, 191

ToS Type of Service. 11, 81

TX Transmission. 16

UAM Universidad Autónoma de Madrid. 178, 190

UDP User Datagram Protocol. 11, 12, 25, 27, 46, 125, 126, 137, 149, 151, 155, 157, 161, 162, 164

UDTS Uniform Distribution of TimeStamp. 6, 67, 69, 72, 73

UMTS Universal Mobile Telecommunications System. 128

UPNA Universidad Pública de Navarra. 178, 190

USB Universal Serial Bus. 13

VLAN Virtual Local Area Network. 11, 122

VoIP Voice over IP. viii, 3, 7, 25, 36, 37, 64, 117–122, 124, 125, 130–132, 135, 137, 141–144, 146–148, 167–169, 176, 177, 179, 188, 189, 191

WDTS Weighted Distribution of TimeStamp. 6, 68, 69, 72–74

WiMAX Worldwide Interoperability for Microwave Access. 128

WLAN Wireless Local Area Network. 128, 129

Chapter 1

Introduction

This chapter provides an overview of this Ph.D. thesis and introduces its motivation, presents its objectives, and finally describes its main contributions outlining its organization.

1.1 Overview and Motivation

Leveraging on the widespread availability of broadband access, the Internet has opened new avenues for information accessing and sharing in a variety of media formats. Such popularity has resulted in an increase of the amount of resources consumed in backbone links, whose capacities have witnessed numerous upgrades to cope with the ever-increasing demand for bandwidth. In addition, the Internet customers have obtained a strong position in the market, which has forced network operators to invest large amounts of money in traffic monitoring on attempts to guarantee the satisfaction of their customers—which may eventually imply a growth in operators' market share. Thus, network monitoring has undoubtedly become a key task for network operators due to such ever-increasing users' demand.

Nevertheless, keeping pace with such ever-increasing data transmission rates is a very demanding task, even if the applications built on top of a monitoring system solely capture to disk the headers of the traversing packets, without further processing them. For instance, traffic monitoring at

rates ranging from 100 Mb/s to 1 Gb/s was considered very challenging a few years ago, whereas contemporary commercial routers typically feature 10 Gb/s interfaces, reaching line-rate capacities of up to Tb/s by means of optical network technologies [SdRHA+10]. Thus, the heterogeneity and high complexity of current networks entails a large number of different applications and protocols, aggregates of multi-10Gb/s worth of traffic, tens of Million packets per second (Mpps) and millions of concurrent flows per link [YZ10]. To cope with such constraints requires high-performance solutions along with scalable and flexible designing that allows processing at line-rate different network data and granularities (packet-level, flow-level, aggregated statistics) with different purposes (e.g., anomaly detection and traffic classification) simultaneously.

Off-The-Shelf (OTS) systems, which are based on commodity hardware and open-source software, have been proven to be an actual alternative to specialized hardware (such as Field-Programmable Gate Array (FPGA)s and network processors), in high-performance computing [Ker12]. Particularly, the research community has presented in the last few years several software solutions based on commodity hardware to perform some specific network traffic monitoring tasks with astonishing results [BDKC10]. Leveraging on OTS systems to build network monitoring applications brings along several advantages when compared to commercial solutions, among which overhang the flexibility to adapt any network operation and management tasks (as well as to make the network maintenance easier), and the economies of scale of large-volume manufacturing in the PC-based ecosystem, ergo entailing large reductions of the Operational Expenditures (OPEX) and Capital Expenditures (CAPEX) investments, respectively. Furthermore, the utilization of commodity hardware presents other advantages such as using energy-saving policies already implemented in PCs [GDMnR+12], and better availability of hardware/software updates that enhances extensibility [HJPM10].

Traffic classification technology has gained importance in the recent years, as it has proven useful in tasks such as accounting, security, service differentiation policies, network design and research [DPC12]. Since its inception up to date, the research community has paid special attention to new ap-

proaches to improve the accuracy of this technology, but it has not been until recently when the evaluation of their performance has gained relevance. Thus, some of the most accurate mechanisms have seen that their execution on high-speed networks is barely improbable. This has increased the interest on mechanisms to reduce the application burden that classifying requires. Such mechanisms are essentially Deep Packet Inspection (DPI) and Machine Learning (ML) tools [NA08], once port-based classification has been ruled out because the widespread use of random port numbers by Peer-to-Peer (P2P) and Voice over IP (VoIP) applications.

Among the diversity of services that travel through Internet, multimedia traffic and, particularly, VoIP is worthy to be highlighted due to its relevance and popularity. For this reason, it has received much attention by the research community [KP09, BMPR10]. VoIP requires a detailed monitoring of the users' Quality of Service (QoS) and Quality of Experience (QoE) to a greater extent than in traditional Public Switch Telephony Network (PSTN)s. As previously shown, such monitoring process must be able to track VoIP traffic in high speed networks, nowadays typically of multi-Gb/s rates. In this case, recent government directives require that providers retain certain information from their users' calls [SGI08]. Similarly, the convergence of data and voice services allows operator to provide new services such as full data retention, in which users' calls can be recorded for either quality assessment (call-centers, QoE), or security purposes (lawful interception). This implies a significant investment on infrastructure, especially on large-scale networks that require multiple points of measurements, given that traffic monitoring tasks are very demanding in terms of computational power.

Thus, we have to face the following challenges:

(i) *High-performance*: coping with line-rate in 10Gigabit Ethernet (GbE) links involves processing speeds of up tens of Mpps and Million flows per second (Mfps).

(ii) *Limited Cost*: guaranteeing large reductions of the OPEX and CAPEX investments.

(iii) *Accuracy*: classifying and identifying network traffic with significant

precision and accuracy.

(iv) *Scalability*: making it the most of parallelism of OTS systems with multi-core architectures and Network Interface Card (NIC)s.

(v) *Flexibility*: monitoring with different tasks simultaneously and processing traffic from the same network in different granularities without additional overhead (running on the same PC, processing data without unnecessary copies).

Consequently, this Ph.D. thesis focuses on overcoming the abovementioned challenges.

1.2 Objectives

The main objective of our work is *to show the feasibility of network traffic monitoring tasks and, particularly, of Internet traffic classification, at line-rate on 10 Gb/s links using OTS systems.* That is, we should be able to answer some questions, such as: *Is an OTS system enough for traffic classification in a highly utilized 10 Gb/s link? Which monitoring tasks are feasible to be performed on an OTS system at wire-speed? Does it scale to 40 Gb/s? Where are the limits of the different processes of a monitoring system: sniffing, timestamping, flow handling, packet and flow classification, and data retention?*

Figure 1.1 shows the general architecture of a networking monitoring system. First, the sniffing module captures traffic from the network, fetching the packets from the NIC to the system memory. This "simple" task has become a challenge due to the ever-increasing data rate of links. For instance, a 10 Gb/s link may carry more than 14 million packets per second. Chapter 3 is devoted to evaluate this module: presentation of current limitations and the general solutions to overcome them; and description, quantitative/qualitative evaluation and comparison of the proposed capture engines.

Along with captured packets is desirable to get accurate timestamps—i.e., the date and time of day when a packet was received. Software (SW)

timestamping requires system calls to obtain the corresponding timestamps requesting to the system clock—whereas Hardware (HW) timestamping is only possible using specialized HW. Note that monitoring a 10 Gb/s link implies interarrival times less than 1 μs, and consequently, to achieve such accuracy and precision of timestamp is actually a challenging task. In Chapter 4, we thoroughly analyze timestamping issues.

Once a packet is received, the capture engine makes it available to upper layers. Many networking applications work at flow-level (aggregation of packets which share determined characteristics, generally, the 5-tuple). Thus, an intermediate flow manager is indispensable to subsequent monitoring tasks. Such flow manager requires to match each packet to the correct flow bin. Chapter 5 studies the performance of this module.

Finally, the different network monitoring functionalities are executed in user application, which are fed by previous modules: either directly reading packets from the capture engine or processing the output of flow manager. One of the most important network monitoring applications is traffic classification, i.e., to associate traffic flows or packets with the corresponding application (or with the application type) that generated them. Traffic classification technologies are used in service differentiation, e.g., to enforce QoS; with security purposes, e.g., for legal interception or for intrusion detection; and for many other uses, such as network design and management, accounting and billing. Especially, multimedia traffic classification and management has awakened interest in both the research community and network operators. Thus, in Chapter 5 we propose and evaluate two architectures for traffic classification and traffic monitoring, while Chapter 6 focuses on multimedia traffic.

1.3 Thesis Structure

First, Chapter 2 describes the state of the art. The first section provides the required background to understand the possibilities that contemporary commodity OTS systems provide for high-performance networking tasks, whereas the second section presents the network monitoring proposals found in the

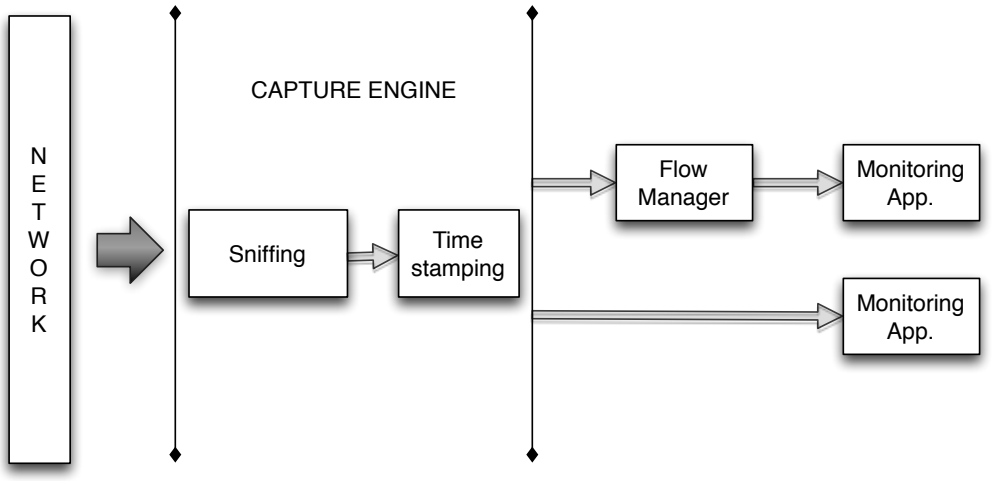


Figure 1.1: General architecture of a network monitoring system

literature built on the basis of OTS systems. The last section is devoted to survey the different approaches proposed in the literature for Internet traffic classification, providing a taxonomy for categorizing the surveyed techniques.

Chapter 3 and 4 analyze two essential aspects for every traffic monitoring system, namely: packet sniffing and timestamping. On the one hand, Chapter 3 presents the limitations of packet sniffing using OTS systems and the general solutions proposed to overcome such limitations. Then, we thoroughly evaluate and compare the different Input/Output (I/O) packet capture engines available as open-source. Finally, we discuss the performance evaluation results, highlight the advantages and drawbacks of each capture engine and give guidelines to the research community in order to choose the more suitable capture system.

On the other hand, Chapter 4 analyzes the accuracy on packet timestamping when using such novel packet I/O engines. We describe the impact of batch processing (such technique is widely implemented in novel packet capture engines) and propose two different approaches to mitigate such impact: (i) Uniform Distribution of TimeStamp (UDTS) and Weighted Distribution of TimeStamp (WDTS) algorithms that distribute the inter-batch time gap among the different packets composing a batch; (ii) a redesign of the net-

work driver, Kernel-level Polling Thread (KPT), to implement a kernel-level thread which constantly polls the NIC buffers for incoming packets and then timestamps and copies them into a kernel buffer one-by-one. Finally, we quantify the timestamp accuracy of batch-based techniques and of the proposed improvements through real experiments using both synthetic traffic and real packet traces.

Chapter 5 evaluates the performance of higher-level modules, such as a flow manager or a traffic classifier, and analyzes the feasibility of line-rate traffic monitoring and classification using OTS systems. In the first part of the chapter, we present an open-software-based traffic classification engine running on commodity multi-core hardware. We perform a thorough sensitivity analysis involving important aspects of the system, such as the use of specific hash functions and efficient data structures for flow management, as well as the use of multiple state-of-the-art machine learning tools. In the second part of the chapter, we propose a modular architecture system, which is able to obtain and process network traces from different levels and granularities at wire-speed. To show the applicability of our system, we present a network traffic monitoring tool, *DetectPro*, implemented over the proposed architecture, which is able to monitor providing statistics, report alarms and afterwards perform forensic analysis, based on packet-level traces, flow-level registers and aggregate statistic logs.

Chapter 6 focuses on the analysis, management and classification of one of the most popular traffic class on the Internet, namely, multimedia traffic. First, we present an overview of multimedia traffic, particularly, VoIP and Skype traffic. Then, we propose two different multimedia traffic manager OTS systems. On the one hand, we present a system, called *RTPTracker*, which is able to (i) capture traffic at multi-Gb/s rates, (ii) identify and track of VoIP traffic, (iii) generate the statistics required to ensure users' QoS as well as in compliance with data retention directives and, (iv) reconstruct and index the VoIP calls to provide novel services based on call recording. On the other hand, we propose a Skype traffic classifier, called *Skypeness*, based on three statistical characteristics of Skype traffic, namely, delimited packet size, nearly constant packet interarrival times and bounded bitrate.

In addition, we assess the impact of packet sampling on traffic classification. Such analysis shows that sampling is not a definitive drawback to identify Skype traffic at multi-10Gb/s rates.

Finally, Chapter 7 concludes this thesis with the main contributions, as well as the industrial applications of the results, and outlines future steps continuing the work presented.

Chapter 2

State of the Art

This chapter provides the required background in the topics related with this thesis. The organization of the chapter is as follows. Section 2.1 presents the possibilities that OTS systems, based on commodity hardware and open-source software, offer to develop network monitoring tasks, and particularly, Internet traffic classification systems. Then, Section 2.2 shows the applicability of OTS systems using as example a set of different network traffic processing and monitoring systems proposed in the literature. Finally, Section 2.3 describes the different techniques to classify and identify Internet traffic, providing a taxonomy of them. In any case, a more detailed revision of the related work of these and different topics will be presented in the corresponding chapters when required.

2.1 Off-The-Shelf Systems: Commodity Hardware and Open-Source Software

2.1.1 Introduction

OTS systems are characterized by sharing a base instruction set and architecture (memory, I/O map and expansion capability) common to many different models, containing industry-standard Peripheral Component Interconnect (PCI) slots for expansion that enables a high degree of mechani-

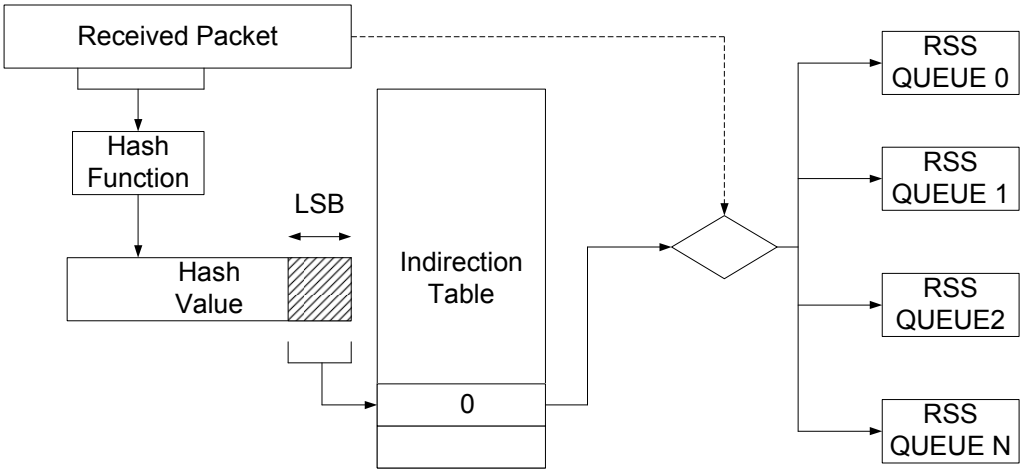


Figure 2.1: RSS architecture

cal compatibility, and whose software is widely available off-the-self. These characteristics play a special role in the economies of scale of the commodity computer ecosystem, allowing large-volume manufacturing with low costs per unit. Furthermore, with the recent development of multi-core Central Processing Unit (CPU)s and off-the-self NICs, these computers may be used to capture and process network traffic at near wire-speed with little or no packet losses in 10 GbE networks [HJPM10].

2.1.2 Novel Network Interface Cards

Modern NICs have evolved significantly in the recent years calling both former capturing paradigms and hardware designs into question. One example of this evolution is Receive Side Scaling (RSS) technology developed by Intel [Int12] and Microsoft [Mic13]. RSS allows NICs to distribute the network traffic load among different cores of a multi-core system, overcoming the bottleneck produced by single-core based processing and optimizing cache utilization. Specifically, RSS distributes traffic to different receive queues by means of a hash value, calculated over certain configurable fields of received packets and an indirection table. Each receive queue may be bound to different cores, thus balancing load across system resources.

Algorithm 1 Toeplitz standard algorithm

```
1: function ComputeHash(input[],K)
2:     result = 0
3:     for each bit b in input[] from left to right do
4:         if b == 1 then
5:             result^= left-most 32 bits of K
6:         shift K left 1 bit position
7:     return result
```

As shown in Figure 2.1, the Least Significant Bits (LSB) from the calculated hash are used as a key to access to an indirection table position. Such indirection table contains values used to assign the received data to a specific processing core. The standard hash function is a Toeplitz hash whose pseudocode is showed in Algorithm 1. The inputs for the function are: an array with the data to hash and a secret 40-byte key (K)—essentially a bitmask. The data array involves the following fields: IPv4/IPv6 source and destination addresses; Transmission Control Protocol (TCP)/User Datagram Protocol (UDP) source and destination ports; and, optionally, IPv6 extension headers. The default secret key produces a hash that distributes traffic to each queue maintaining unidirectional flow-level coherency—packets containing same source and destination addresses and source and destination ports will be delivered to the same processing core. Modifying the secret key to distribute traffic based on other features can change this behavior. For example, in [WP12] a solution for maintaining bidirectional flow-level (session-level) coherency is shown.

Modern NICs offer further features in addition to RSS technology. For example, advanced hardware filters can be programmed in Intel 10 GbE cards to distribute traffic to different cores based on rules. This functionality is called Flow Director and allows the NIC to filter packets by: Source/destination addresses and ports; Type of Service (ToS) value from Internet Protocol (IP) header; Level 3 and 4 protocols; and, Virtual Local Area Network (VLAN) value and Ethertype.

Another important functionality that Intel 10GbE cards offer is high precision hardware timestamping for its use in synchronization Precision Time

Protocol (PTP) systems [IEE08]. This feature allows user-space synchronization utilities to access to the value of accurate timestamps for a filtered set of defined PTP packets by means of hardware register reads. The filtering process takes into account the UDP packet port number and PTP message identifier, and is programmed via register writes. Note that this feature does not allow timestamping every packet—only certain PTP packets by default.

2.1.3 Non Uniform Memory Access Systems

The number of CPU cores within a single processor is continuously increasing, and nowadays it is common to find quad-core processors in commodity computers—and even several eight-core processors in commodity servers.

HW/SW interactions are also of paramount importance in commodity hardware systems. For example, Non Uniform Memory Access (NUMA) design has become the reference for multiprocessor architectures, and has been extensively used in high-speed traffic capturing and processing. In more detail, NUMA design splits available system memory between different Symmetric MultiProcessor (SMP) assigning a memory chunk to each of them. The combination of a processor and a memory chunk is called NUMA node. Figure 2.2 shows some examples of NUMA architectures. NUMA memory distribution boosts up systems' performance as each processor can access in parallel to its own chunk of memory, reducing the CPU data starvation problem. Although NUMA architectures increase the performance in terms of cache misses and memory accesses [DAR12], processes must be carefully scheduled to use the memory owned by the core in which they are being executed, avoiding accessing to other NUMA nodes.

Essentially, the accesses from a core to its corresponding memory chunk results in a low data fetching latency, whereas accessing to other memory chunk increases this latency. To explode NUMA architectures, the NUMA node distribution must be previously known as it varies across different hardware platforms. Using the `numactl`[1] utility a NUMA node distance matrix may be obtained. This matrix represents the distance from each NUMA node

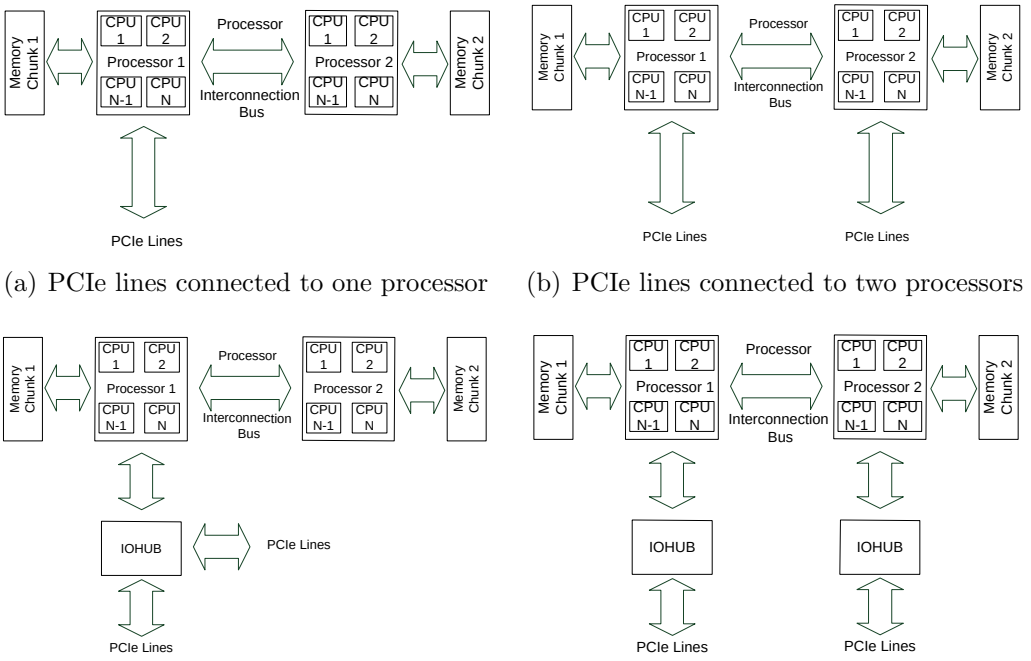
[1] `linux.die.net/man/8/numactl`

memory bank to the others. Thus, the higher the distance is, the higher the access latency to other NUMA nodes is. Other key aspect to get the most of NUMA systems is the interconnection between the different devices and the processors.

Generally, in a traffic capture scenario, NICs are connected to processors by means of PCI-Express (PCIe) buses. Depending on the used motherboard in the commodity hardware capture system, several interconnection patterns are possible. Figure 2.2 shows the most likely to find schemes on actual motherboards. Specifically, Figure 2.2(a) shows an asymmetric architecture with all PCIe lines directly connected to a processor whereas Figure 2.2(b) shows a symmetric scheme where PCIe lines are distributed among two processors. Figures 2.2(c) and 2.2(d) show similar architectures with the difference of having their PCIe lines connected to one or several I/O Hub (IOH). IOH not only connect PCIe buses but also Universal Serial Bus (USB) or PCI buses as well as other devices with the consequent problem of sharing the bus between the IO-hub and the processor among different devices. All this aspects must be taken into account when setting up a capture system. For example, when a NIC is connected to PCIe assigned to a NUMA node, capturing threads must be executed on the corresponding cores of that NUMA node. Assigning capture threads to another NUMA node implies data transmission between processors using Processor Interconnection Bus which leads to performance degradation. One important implication of this fact is that having more capture threads than existing cores in a NUMA node may be not a good approach as data transmission between processors will exist. To obtain information about the assignment of a PCIe device to a processor, the following command can be executed on Linux systems `cat /sys/bus/pci/devices/PCI_ID/local_cpulist` where PCI_ID is the device identifier obtained by executing `lspci`[2] command.

Recently new high-performance processing paradigms have arisen to the commodity hardware scenario. One example of these technologies are the Graphic Processing Unit (GPU)s. GPUs, traditionally used to render and process images and video, are being used to process all types of data. GPUs

[2] `linux.die.net/man/8/lspci`



(a) PCIe lines connected to one processor

(b) PCIe lines connected to two processors

(c) PCIe lines connected to one IOHUB

(d) PCIe lines connected to two IOHUB

Figure 2.2: NUMA architectures

allow the processing of large amounts of data efficiently making parallel use of numerous specialized cores. Such cores are simpler than traditional CPU cores but are specialized on mathematical and logical operations executed in parallel over a given set of data. Additionally, GPUs are equipped with efficient memory hierarchies that assure low latency accesses and high performance data transfers. Modern high-speed processing systems are taking advantage of these characteristics combining traditional multi-core programming with this new paradigm to speedup different applications. Specifically, on the field of high-speed packet processing several approaches have used GPUs to speedup tasks such as IP packets lookup/routing, packet encryption or traffic analysis [HJPM10, VPI11].

All the previously mentioned characteristics make modern commodity computers highly attractive for high-speed network traffic monitoring, because their performance may be compared to today's specialized hardware, such as FPGAs (NetFPGA [Uni13], Endace DAG cards [End12]), network

processors [AL13, LSI12, Int13] or commercial solutions provided by router vendors [Cis09], but they can be obtained at significantly lower prices, thus providing cost-aware solutions. Moreover, as the monitoring functionality is developed at user-level, commodity hardware-based solutions are largely flexible, which in addition to the mechanical compatibility, allows designing scalable and extensible systems that are of paramount importance for the monitoring of large-scale networks.

2.1.4 Operating System Network Stack

Nowadays, network hardware is rapidly evolving for high-speed packet capturing but software is not following this trend. In fact, most commonly used operating systems provide a general network stack that prioritizes compatibility rather than performance. Modern operating systems feature a complete network stack that is in charge of providing a simple socket user-level interface for sending/receiving data and handling a wide variety of protocols and hardware. However, this interface does not perform optimally when trying to capture traffic at high speed.

Specifically, Linux network stack in kernels previous to 2.6 followed an interrupt-driven basis. Let us explain its behavior: each time a new packet arrives into the corresponding NIC, this packet is attached to a descriptor in a NIC's Receiving (RX) queue. Such queues are typically circular and are referred as rings. This packet descriptor contains information regarding the memory region address where the incoming packet will be copied via a Direct Memory Access (DMA) transfer. When it comes to packet transmission, the DMA transfers are made in the opposite direction and the interrupt line is raised once such transfer has been completed so new packets can be transmitted. This mechanism is shared by all the different packet I/O existing solutions using commodity hardware. The way in which the traditional Linux network stack works is shown in Figure 2.3. Each time a packet RX interrupt is raised, the corresponding interrupt software routine is launched and copies the packet from the memory area in which the DMA transfer left the packet, DMA-able memory region, into a local kernel `sk_buff` structure—typically,

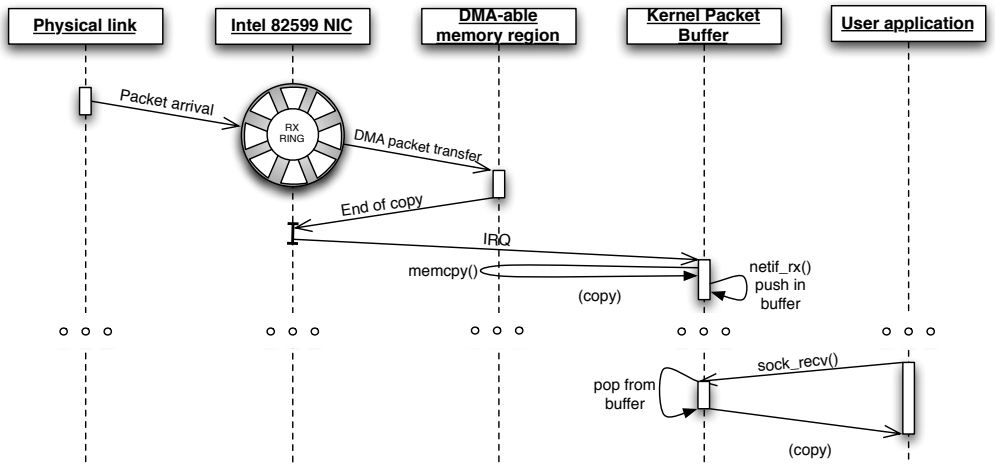


Figure 2.3: Linux Network Stack RX scheme in kernels previous to 2.6

referred as packet kernel buffer. Once that copy is made, the corresponding packet descriptor is released (then the NIC can use it to receive new packets) and the `sk_buff` structure with the just received packet data is pushed into the system network stack so that user applications can feed from it. The key point in such packet I/O scheme is the need to raise an interrupt every time a packet is received or transferred, thus overloading the host system when the network load is high [ZFP12].

With the aim of overcoming such problem, most current high-speed network drivers make use of the New API (NAPI) approach [Fou12]. This feature was incorporated in kernel 2.6 to improve packet processing on high-speed environments. NAPI contributes to packet capture speedup following two principles:

(i) *Interrupt mitigation.* Receiving traffic at high speed using the traditional scheme generates numerous interrupts per second. Handling these interrupts might lead to a processor overload and therefore performance degradation. To deal with this problem, when a packet RX/Transmission (TX) interrupt arrives, the NAPI-aware driver interrupt routine is launched but, differently from the traditional approach, instead of directly copying and queuing the packet the interrupt routine schedules the execu-

tion of a `poll()` function, and disables future similar interrupts. Such function will check if there are any packets available, and copies and enqueues them into the network stack if ready, without waiting to an interruption. After that, the same `poll()` function will reschedule itself to be executed in a short future (that is, without waiting to an interruption) until no more packets are available. If such condition is met, the corresponding packet interrupt is activated again. Polling mode is more CPU consumer than interrupt-driven when the network load is low, but its efficiency increases as speed grows. NAPI compliant drivers adapt themselves to the network load to increase performance on each situation dynamically. Such behavior is represented in Figure 2.4.

(ii) *Packet throttling.* Whenever high-speed traffic overwhelms the system capacity, packets must be dropped. Previous non-NAPI drivers dropped these packets in kernel-level, wasting efforts in communication and copies between drivers and kernel. NAPI compliant drivers can drop traffic in the network adapter by means of flow-control mechanisms, avoiding unnecessary work.

From now on, the GNU Linux NAPI mechanism will be used as the leading example to illustrate the performance problems and limitations as it is a widely used open-source operating system which makes performance analysis easier and code instrumentation possible for timing statistics gathering. Furthermore, the majority of the existing proposals in the literature are tailored to different flavors of the GNU Linux distribution. Some of these proposals have additionally paid attention to other operating systems, for example, FreeBSD [Riz12a], but none of them have ignored GNU Linux.

2.1.5 Conclusion

The utilization of OTS systems, based on commodity HW and open-source SW, may be a great alternative to specialized hardware, performing high-performance computing and networking tasks. First, modern NICs have different capabilities, such as RSS multi-queue packet RX, advanced packet

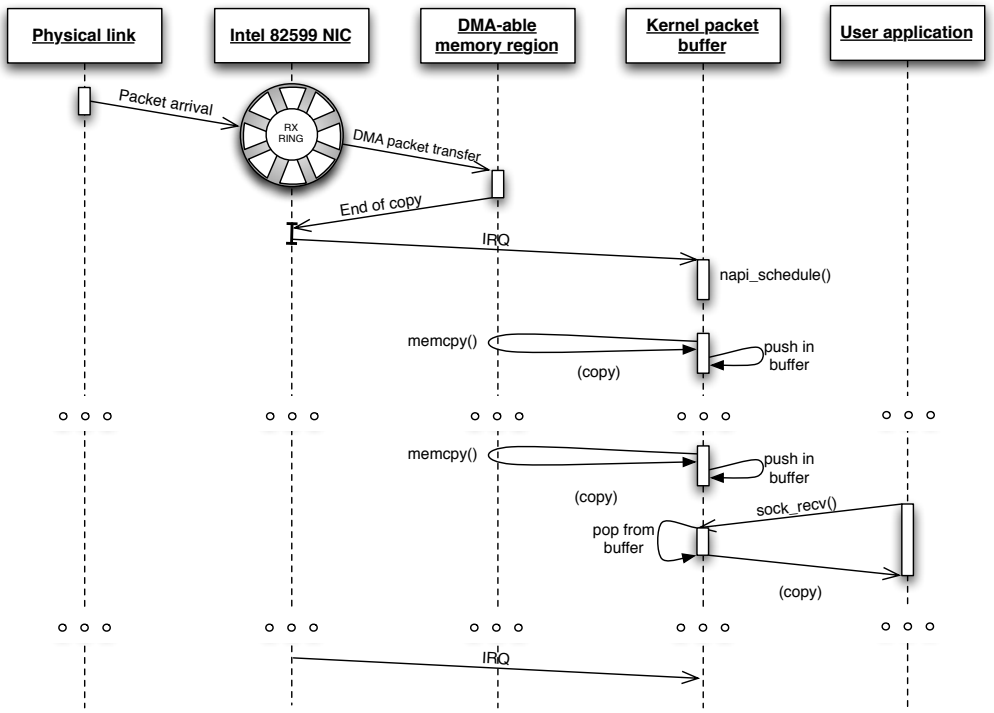


Figure 2.4: Linux NAPI RX scheme

filtering and distributing and high precision HW packet timestamping. On the other hand, current commodity hardware, thanks to the use of multi-core processors and NUMA systems, may achieve high-performance at low cost. Finally, modern operating systems are not following the rapid evolution of network hardware for high-speed packet sniffing due to general-purpose-aware (instead of high-performance-aware) design .

In Chapter 3, we identify the limitations of modern operating systems, describe the proposed general improvements to overcome such limitations, evaluate and compare each particular proposed capture engine. Chapter 4 analyzes another essential feature of a monitoring system when using such novel packet capturing engines.

2.2 High-Performance Traffic Processing Systems

2.2.1 Introduction

In this section, we turn our attention to the challenges that network application developers face when using OTS systems. Essentially, they face these two challenges: contrary to traditional network applications, these new engines potentially extend the capture data rates by several orders of magnitude. This may imply that current networking software is not able to adequately work at this speed. On the other hand, multi-core hardware opens a new opportunity to develop applications taking advantage of the parallelism that such engines allow, however current applications were developed in the pre-multicore era. In this section, we present examples of the limitations and the proposed solutions that networking application developers have suggested to deal with these two challenges. For instance, complex and versatile monitoring platforms have been proposed in the literature. One example of such a network monitoring tool is Tstat [FMM+11]. Tstat is an open source tool that gives researchers and network operators live and flexible monitoring in a modular and versatile fashion. Tstat software runs on commodity hardware, but only achieves Gb/s rates and it is not able to cope with current and future high-speed networks of 10 Gb/s and beyond.

Nevertheless, in recent years, we have found in the literature examples whose results are astonishing given the cost of the final systems. There are several proposals that implement specific monitoring tasks such as traffic classification [GES12], Network Intrusion Detection System (NIDS) [JLM+12, VPI11] or the implementation of software routers, which are able to process traffic at 10Gb/s rates. However, such systems follow a monolithic approach (do not allow being simultaneously executed along with other monitoring application over the same interface) and lack of the required flexibility in a high-speed network. That is, the possibility of network traffic monitoring at different granularities (packet-level, flow-level, aggregated statistics) with different purposes (e.g., anomaly detection and traffic classification) simulta-

neously. Table 2.1 summarizes the performance and characteristics of some of these applications.

2.2.2 Flow Matching

As second step, after sniffing packets, monitoring (at the flow level) requires to match each packet to the correct flow bin. In software-based solutions such as Tstat [RM06] or YAF [IT10] this is usually accomplished using hash-based structures over the flow 5-tuple. To the best of our knowledge, however, performance of flow matching code in complex monitoring systems is rarely evaluated alone and extrapolating such data from overall measurements can be tough or even misleading. For instance, [IT10] describes a flow management module in detail, explaining how to optimize flow management using slab allocator [Bon94] for fast recycling of expired flow records, but benchmarks of the system performance are not publicly available. Otherwise, the performance analysis for flow matching modules has been done either monitoring real Internet Service Provider (ISP) deployments [FMM+11] or over off-line traces [RM06, WXD11]. However, as real 10 Gb/s traffic is not by itself a stress-test scenario, this calls for synthetic benchmarks.

Explicit performance are reported instead in [Der08] where a dual Xeon box hosts a dedicate Endace DAG card which achieves matching of up to 6 Mpps. In [QXH+07] an Intel IXP2850 Network Processor is shown matching 10 million concurrent flows at 10 Gb/s at full packet rate. Switching to off-the-shelf setup, an application note from Intel [LK09] reports flow matching of trains of 64 bytes packets at 17 Mpps out of 24 Mpps received over 16× 1 Gb/s interfaces, where each NIC is tied to a different core of an Intel multi-core CPU system (unfortunately the study does not report the number of concurrent flows). A similar architecture [DDDS11] matches up to 11 Mpps for 1 million concurrent flows at 10 Gb/s using "FastFlow" algorithms spawned over 6 cores.

2.2.3 Software Routers

The use of commodity hardware to perform high-speed tasks started with the significant increase in popularity that software routers have achieved in the last years. Software routers present some interesting advantages with respect to hardware-design routers—essentially cost and flexibility. This increase has been strengthened by multiple examples of successful implementations, and by the appearance of GPUs which multiply the parallelism between processes while the cost remains low.

The authors in [HJPM10] developed a software router, called PacketShader, able to work at multi-10Gb/s rates. To this end, they proposed to move the routing process from the CPU to GPUs, where hundreds of threads can be executed in parallel. As most software routers operate on packet headers, the use of GPUs and parallel threads fits adequately. Therefore, it is intuitive to bind each received packet to a thread in a GPU, which multiplies the capacity of the router by the number of concurrent threads in each GPU. The results are astonishing given the use of commodity hardware and software solutions. Specifically, IPv4 forwarding achieves throughputs of 39 Gb/s with 64B packets, and even better results for larger packet sizes in a unique machine. Specifically, this is based on two quad-core processors of 2.66 GHz, whose cost together with the rest of the required hardware is about $7000 (in 2010). The results for IPv6 forwarding are only lightly below—38 Gb/s. In addition to a software router, the authors also evaluated the performance of their approach working as an OpenFlow[MAB+08] Switch and an Internet Protocol Security (IPSec) [FK11] gateway. The results show that they are able to switch at 32 Gb/s, and they obtained a throughput of 10.2 Gb/s for IPSec overcoming commercial solutions.

Similarly to PacketShader's approach, the authors in [RCC12] proposed netmap. They evaluated the performance of a router software, specifically a Click Modular Router developed years ago [KMC+00]. Conversely to PacketShader system, they did not use GPUs to parallelize tasks, thus the performance at application-level was lower, about 2 Gb/s at 64B packets, although the capture engine showed similar behavior. However, the authors found a

roadblock in their study that deserves to be remarked. Essentially, Click was not developed to support such a data rate and it required to be optimized. This was the second challenge that we pointed out at the beginning of this section: old implementations must be reviewed to work properly with these new capture systems. In this case, the authors pinpointed that the process of allocating memory in the C++ code of Clicks was not ideal. In the original version, two blocks of memory were reserved per packet—one for the payload and another for its descriptor. However, this was not necessary as the memory can be recycled inside the code, avoiding the allocation of new one, and using fixed-size objects. The improvement ranges between 3x and 4x depending on the size of the batches, which represents a significant gain. This result alerts us that some old implementations of popular networking applications have been overtaken by the capture engines, which calls for a review and subsequent optimization of such software. Specifically, from the latter article we have identified that similarly to the case of lower levels, the allocation of memory in a per-packet basis at application-level is neither acceptable. Instead, pre-allocation of the required memory and some periodically processes to increase or free (as a garbage collector) memory are necessary to achieve high speed rates.

2.2.4 NIDS: Network Intrusion Detection Systems

Network intrusion detection is one of the most important tasks to be carried out by monitoring systems, given its importance in network security. Essentially, there are two approaches to implement NIDS: those based on characteristics of the traffic and those related to DPI, which basically consists in searching for a given signature in the traffic payload. While the former typically allows the achievement of faster speeds, the use of DPI approaches tends to be more accurate.

The authors in [VPI11] evaluated this latter option proposing a full software implementation, called MIDeA, based on multi-core commodity hardware and GPUs, similar to the previously explained PacketShader. In a nutshell, they take advantage of the parallelism of current NICs and CPU

Table 2.1: Summary of the performance and characteristics of a set of typical high-performance network applications using commodity hardware

System Name	Application	Throughput
PacketShader [HJPM10]	IPv4 forwarding	39 Gb/s
	IPv6 forwarding	38 Gb/s
	OpenFlow Switch	32 Gb/s
	IPSec gateway	10.2 Gb/s
MIDeA [VPI11]	NID system	< 7.5 Gb/s
Szabó et al. [SGV+10]	Traffic classification (DPI, connection pattern, port based)	6.7 Gb/s

cores, as well as GPUs. They present a prototype implementation of a NIDS based on Snort [CR04], the de facto standard software for this purpose, which includes more than 8,192 rules and 193,000 strings for string matching purposes. The results show that their system, whose architecture is composed of two processors of four cores each one at 2.27 GHz and with a value of $2739 (in 2011), is able to achieve 7.22 Gb/s for synthetic traces in the ideal scenario of packets of 1500 bytes. This represents an improvement of more than 250% over traditional multi-core implementations. However, the performance remains below 2 Gb/s in the case of packet sizes of 200 bytes. While this presents a significant cut, it is worth noticing that the average Internet packet size is clearly larger than such 200 bytes. In fact, when the system is evaluated with real traces, it achieves rates of 5.7 Gb/s.

2.2.5 Conclusion

The utilization of OTS systems in high-performance tasks, previously reserved to specialized hardware, has raised great expectation in the Internet community, given the astonishing results that some approaches have attained at low cost. We have shown a set of different applications, such as flow matching, SW routing and NIDS, as cases of success in the use of OTS. This makes this solution a promising technology at the present and future, although the

proposed solutions usually follow a monolithic approach and may lack of flexibility and scalability.

In Chapter 5, we analyze the feasibility of flexible network monitoring and, particularly, Internet traffic classification, for high-performance and OTS systems. Chapter 6 focuses on the case of multimedia traffic, taking as example both VoIP tracking and Skype traffic identification.

2.3 Internet Traffic Classification

2.3.1 Introduction

One of the most relevant network monitoring task is traffic classification. Thus, the ability to identify which application is generating every single traffic session is recognized as a crucial building block of today IP networks and unavoidable requisite for their evolution [DPC12]. Effective techniques could open new possibilities for actual deployment of QoS, for enforcing user traffic to comply with policies, for legal interception and intrusion detection [KCF+08].

The latest years have also seen a flurry of proposals exploiting different "features" (in machine learning terms) to perform the classification [NA08]. Statistical techniques based on the size and directions of the first few packets of a flow [BTS06, CDGS07] emerged as especially appealing due to their low complexity if compared to current state-of-the-art DPI approaches. Furthermore, such techniques can be used when traffic is encrypted, while DPI approaches simply cannot. However, most of the previous work on statistical classification focused on assessing the accuracy of the different techniques (that we take for granted given results in [BTS06, CDGS07, KCF+08, LKJ+10]) without measuring their achievable classification rates.

2.3.2 Traffic Classifier Taxonomy

In this section, we provide an overview of the different traffic classification techniques. Table 2.2 shows the general taxonomy of Internet traffic classi-

Table 2.2: General taxonomy of traffic classification techniques

Category	Subcategory	Examples
Port-based	Static Dynamic	CoralReef [MKK+01] SIP/RTP Class.[AKA08]
User-behavior-based	-	BLINC [KPF05]
Payload-based	Signature Matching Stochastic Inspection	L7-filter [Lf09] KISS [FMMR10]
Flows-statistics-based	-	[BTS06, CDGS07, LKJ+10]

fiers. In the followings, we describe the different categories.

2.3.3 Port-based: Static and Dynamic

Traditional services and protocols, such as File Transfer Protocol (FTP), web-browsing or Simple Mail Transfer Protocol (SMTP), are not difficult to detect by simple matching to well-known transport layer (TCP/UDP) port numbers. An example of usage of this technique is CoralReef [MKK+01, CAI11]. Such tool provides a list that maps each port number (or range) with an application or service. However, there are many applications, such as P2P-based ones, that use random, unpredictable or obscure port numbers [KBB+04], and, conversely, there are other applications that may use well-know ports of different services —such as Skype, which uses TCP ports 80 or 443 when the communication using UDP or other TCP ports is not possible [BMM+08].

Other applications may be mapped into a given port number, but only in a bounded time interval. That is the case of Real-time Transport Protocol (RTP). In a VoIP context, Session Initiation Protocol (SIP) protocol [RSC+02] uses plain-text messages in combination with Session Description Protocol (SDP) [HJ98] messages to establish and negotiate the parameters of the associated RTP streams, particularly the pot number to use.

Thus, to indetify RTP packets, SIP packets (corresponding to the initialization phase) must be parsed for extracting the port numbers of the subsequent RTP flows.

Consequently, classification systems must inspect other characteristics of network traffic, such as packet payload or flow statistical features, to be able to classify network traffic.

2.3.4 Payload-based: DPI and Stochastic Inspection

Classic techniques based on DPI have been thoroughly analyzed during the years: though specialized hardware based on Network Processor [LXSL08] and FPGAs [MNB07] have been considered, the emergence of multi-core commodity hardware has gained increasing attention and exhaustive performance analyses have been reported also for advanced systems using off-the-shelf GPUs [SGV+10, LXSL08].

In this regard, the authors of [SGV+10] show a flow-traffic classification system (in a PC with two dual-core processors at 2 GHz) whose throughput achieves a rate of 6.7 Gb/s with real traces—packet sizes of approximately 500 B. They apply DPI, connection patterns (i.e., analyzing the interaction in terms of number of connections or ports that the communication between hosts involves) and port-based classification. Again, they take advantage of the parallelism that GPUs provide to improve performance. In this case, the authors pay significant attention to the especial architecture of the GPUs, and design the software bearing this in mind. Specifically, the GPU fast cached memory tends to be too small to allocate the state machines that a traffic classification system requires. Thus, the authors propose to implement such state machines using the Zobrist hashing algorithm [Zob70]. Basically, it reduces the memory requirements of state machines, which enable their allocation in cached memory. One more time, they show that adapting applications to the new capacities of the hardware, in this case GPUs, is an essential step to obtain the best performance. Similarly, [LXSL08] achieves 3.5 Gb/s of aggregated traffic rate, corresponding to less than 2 Mpps.

Thus, despite the powerfulness of the underlying hardware none of afore-

mentioned approaches is able to actually sustain a 10 Gb/s throughput.

In any case, unfortunately, DPI-based techniques applicability is limited [NA08]. Applications that use encryption algorithms to obfuscate their packets' payload cannot be classified by DPI-based tools. Additionally, governments and legal authorities may prohibit inspecting payload by third parties for privacy issues. On the other hand, DPI techniques require to know the syntax of packet payload and maintain updated the corresponding signatures, which is a difficult task in the ever-changing Internet context.

Stochastic Inspection

With the aim of reducing the tedious and difficult work of building and updating DPI signatures, the authors of [FMMR10] propose a method, called Chi-Square Signatures (KISS), which is able to automatically identify the protocol format using statistical packet inspection. The idea behind of KISS is that the very first bytes of an UDP packet probably carry some application layer protocol fields such as counters, constant values or random identifiers. Thanks to apply a Pearson's Chi-Square test on the payload, we can extract the format and characterize the different application protocols.

KISS may be used only over UDP traffic and is ineffective when the packet payload is encrypted . Additionally, KISS may present similar computational limitations as DPI-based method.

2.3.5 User-behavior-based

As an alternative to DPI and port-based techniques, the authors of [KPF05] propose a classification engine, called BLINd Classification (BLINC), based on behavior patterns of the hosts at transport layer. Thus, BLINC is able to classify traffic without inspecting the packet payload and without knowing the used port number for the sessions. Such restrictions fulfill privacy and practical concerns. BLINC analyzes connections in three level, namely: (i) social aspects, which measures popularity of nodes and clusters them into group/communities of clients or servers; (ii) functional level, which identifies producers or consumers (or hosts playing both roles) in the network; and

(iii) application level, which exploits other flow characteristics (such as the average packet size) to refine the classification algorithm.

BLINC accuracy depends on the topological location of the probe (edge vs. backbone). In this light, BLINC presents poor accuracy results when is monitoring a backbone link, even less than 50% [KCF+08].

2.3.6 Statistical Classification

Statistical techniques, unlike previous classifiers, observe basic properties, such as packet length and interarrival times, to classify traffic and have been celebrated for their accuracy and speed [NA08]. However, while the former has been experimentally demonstrated [KCF+08, LKJ+10] on actual traffic traces, the latter is far from being assessed. As a result, the classification rates and scalability of these new algorithms are either unknown or really far from those needed for real world deployment [DPC12]: e.g., [BTS06, CDGS07] merely discuss the complexity of the classification technique, while the most recent performance analysis in [LKJ+10] reports as few as $30{\cdot}10^3$ classification per seconds with Naïve-Bayes. These methods will be throughly analyzed in Chapters 5 and 6. Thus, a more detailed revision of the related work of these techniques will be presented in the corresponding chapters when required.

2.3.7 Conclusion

Classification accuracy has been widely studied by the research community [NA08, KCF+08, LKJ+10], but not is the case of computational performance of such classification techniques. Consequently, Chapters 5 and 6 focus on analyzing the feasibility of these machine learning techniques for on-line classification of network traffic at 10 Gb/s rates using OTS systems.

Chapter 3

High-Performance Packet Sniffing

The first step of any network monitoring system is to sniff packets from the network. This "simple" task has become a challenge due to the ever-increasing data rate of links. For instance, a 10 Gb/s link may carry more than 14 million packets per second. In spite of the great evolution in commodity hardware capabilities, thanks to multi-core processors and multi-queue network cards among other features, networking stacks and drivers of current operating systems are not be able to keep pace with such commodity hardware and waste its potential performance. In this chapter, we detail the limitations of current operating systems for packet sniffing, and describe both the general solutions to overcome such limitations and the particular capture engines proposed in the literature. Then, we thoroughly evaluate and compare such proposals and provide practitioners and researchers with a road map to implement high-performance networking systems in OTS systems.

3.1 Introduction

Nowadays, if we want to analyze traffic from a backbone link, we must be able to perform all tasks of the system (i.e., capturing, timestamping, flow processing, traffic classification and analysis) at 10 Gb/s, or even more 40-

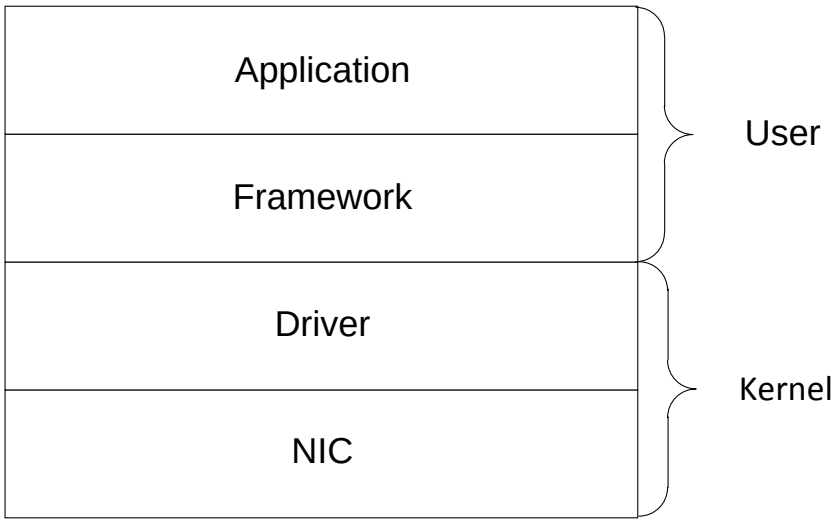


Figure 3.1: Generic structure of a high-speed monitoring OTS system

100 Gb/s in the near future. Particularly, in this chapter, we focus on this first module—essential in any monitoring system.

Figure 3.1 shows the structure of a high-speed monitoring system using commodity hardware, essentially, a 4-layer stack. The first level is the NIC, then, a driver accesses to the NIC and put data at disposal of the operating system kernel (second level). The third level is a set of functions or a framework which enables to applications of user-level access to data, e.g. the de facto packet capture framework libpcap [PCA10]. In general, the combination of a driver and a framework could be called capture engine. Finally, the fourth level is the application level, where the different functionalities are implemented, in our case, monitoring tasks. However, this 4-level structure could change because of certain tools propose taking load from a given level to a lower level with performance purposes.

Thus, to develop a network monitoring solution using commodity OTS systems, the first step is to optimize the default NIC driver to guarantee that the high-speed incoming packet stream is captured lossless. The main prob-

lem, the receive live-lock, was studied and solved several years ago, as shown in 2.1.4. The problem appears when, due to heavy network load, the system collapses because all its resources are destined to serve the per packet interrupts. The solution, which mitigates the interrupts in case of heavy network load, is now implemented in modern operating systems (Section 2.1.4), and specifically in the GNU Linux distribution, which we take in this chapter as the leading example. However, such solution is not enough to cope with 10 Gb/s. Thus, in recent years, several research works have been conducted on this topic. Different particular high-speed packet sniffing solutions have been proposed, which improves the default driver and operating system network stack. In this chapter, we describe the limitations of such default configuration (driver and operating system networking stack), present the general techniques applied to overcome such limitations and compare the different proposed capture engines.

The rest of the chapter is organized as follows: Section 3.2 is devoted to present the limitations of current operating systems for packet sniffing which wastes the potential performance that could be obtained using modern OTS systems. In Section 3.3, we describe the general proposed techniques to overcome the previously shown limitations, whereas in Section 3.4, we detail the different packet capture engines proposed in the literature and qualitatively compare them. Section 3.5 shows the experimental testbed where the performance evaluation analysis of the packet capture engines, presented in Section 3.6, was carried out. Finally, Section 3.7 summarizes the chapter and highlights the main findings and conclusions.

3.2 Packet Sniffing Limitations: Wasting the Potential Performance

NAPI technique by itself is not enough to overcome the challenging task of very high-speed traffic capturing since other architectural inherent problems degrades the performance. After extensive code analysis and performance tests, several main problems have been identified [HJPM10, Riz12a, LZB11,

PVA+12]:

(i) *Per-packet allocation and deallocation of resources.* Every time a packet arrives to a NIC, a packet descriptor is allocated to store packet's information and header. Whenever the packet has been delivered to user-level, its descriptor is released. This process of allocation and deallocation generates a significant overhead in terms of time especially when receiving at high packet rates—as high as 14.88 Mpps in 10 GbE. Additionally, the `sk_buff` data structure is large because it comprises information from many protocols in several layers, when the most of such information is not necessary for numerous networking tasks. As shown in [LZB11], `sk_buff` conversion and allocation consume near 1200 CPU cycles per packet, while buffer release needs 1100 cycles. Indeed, `sk_buff`-related operations consume 63% of the CPU usage in the reception process of a single 64B sized packet [HJPM10].

(ii) *Serialized access to traffic.* Modern NICs include multiple HW RSS queues that can distribute the traffic using a hardware-based hash function applied to the packet 5-tuple (Section 2.1.2). Using this technology, the capture process may be parallelized since each RSS queue can be mapped to a specific core, and as a result the corresponding NAPI thread, which is core-bound, gathers the packets. At this point all the capture process has been parallelized. The problem comes at the upper layers, as the GNU Linux network stack merges all packets at a single point at network and transport layers for their analysis. Figure 3.2 shows the architecture of the standard GNU Linux network stack. Therefore, there are two problems caused by this fact that degrade the system's performance: first, all traffic is merged in a single point, creating a bottleneck; second, a user process is not able to receive traffic from a single RSS queue. Thus, we cannot make the most of parallel capabilities of modern NICs delivered to a specific queue associated with a socket descriptor. This process of serialization when distributing traffic at user-level degrades the system's performance, since the obtained speedup at driver-level is lost. Additionally, merging traffic

from different queues may entail packet disordering [WDC11].

(iii) *Multiple data copies from driver to user-level.* Since packets are transferred by a DMA transaction until they are received from an application in user-level, it turns out that packets are copied several times, at least twice: from the DMA-able memory region in the driver to a packet buffer in kernel-level, and from the kernel packet buffer to the application in user-level. For instance, a single data copy consumes between 500 and 2000 cycles depending on the packet length [LZB11]. Another important idea related to data copy is the fact that copying data packet-by-packet is not efficient, so much the worse when packets are small. This is caused by the constant overhead inserted on each copy operation, giving advantage to large data copies.

(iv) *Kernel-to-userspace context switching.* From the monitoring application in user-level is needed to perform a system call for each packet reception. Each system call implies a context switch, from user-level to kernel-level and vice versa, and the consequent CPU time consumption. Such system calls and context switches may consume up to 1000 CPU cycles per-packet [LZB11].

(v) *No exploitation of memory locality.* The first access to a DMA-able memory region implies cache misses because DMA transactions invalidate cache lines. Such cache misses represent 13.8% out of the total CPU cycles consumed in the reception of a single 64B packet [HJPM10]. Additionally, as previously explained, in a NUMA-based system the latency of a memory access depends on the memory node accessed. Thus, an inefficient memory location may entail performance degradation due to cache misses and greater memory access latencies.

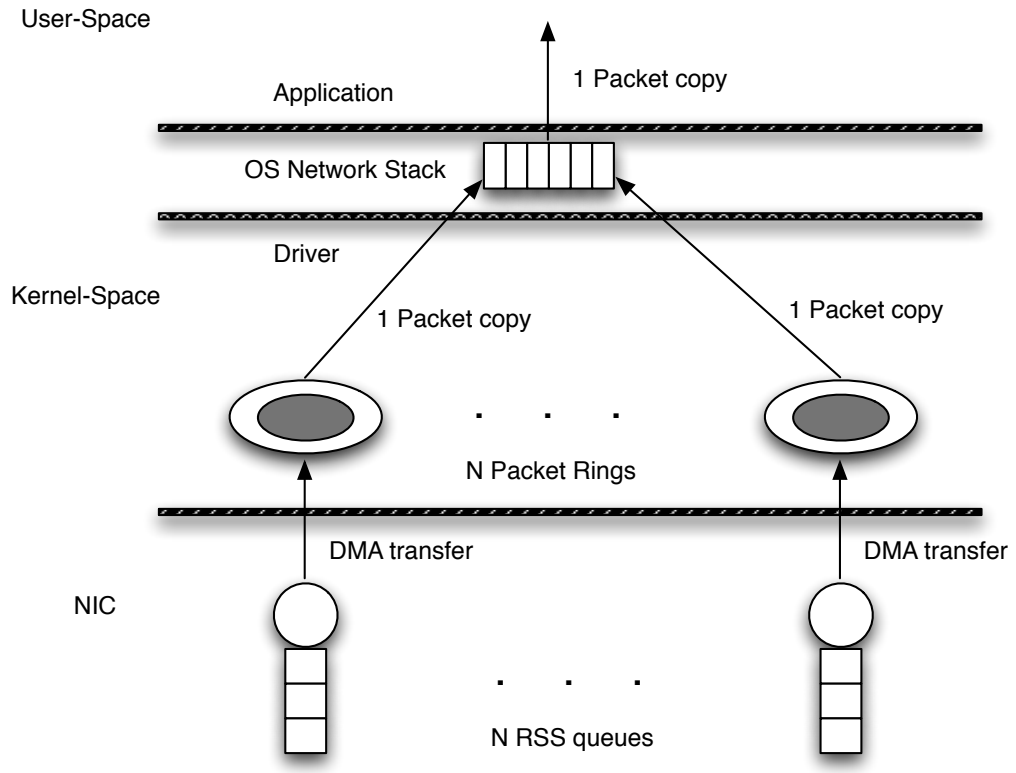


Figure 3.2: Standard Linux Network Stack

3.3 Proposed Techniques to Overcome Limitations

In Section 2.1.2, we have shown that modern NICs are a great alternative to specialized hardware for network traffic processing tasks at high speed. However, as shown in Section 3.2, both the networking stack of current operating systems and applications at user-level do not properly exploit these new features. In this section, we present several proposed techniques to overcome the previous described limitations in the default operating systems' networking stack.

Such techniques may be applied either at driver-level, kernel-level or between kernel-level and user-level, specifically applied at the data they ex-

change, as will be explained.

(i) *Pre-allocation and re-use of memory resources.* This technique consists in allocating all memory resources required to store incoming packets, i.e., data and metadata (packet descriptors), before starting packet reception. Particularly, N rings of descriptors (one per HW queue and device) are allocated when the network driver is loaded. Note that some extra time is needed at driver loading time but per-packet allocation overhead is substantially reduced. Likewise, when a packet has been transferred to user-space, its corresponding packet descriptor is not released to the system, but it is re-used to store new incoming packets. Thanks to this strategy, the bottleneck produced by per-packet allocation/deallocation is removed. Additionally, `sk_buff` data structures may be simplified reducing memory requirements. These techniques must be applied at driver-level.

(ii) *Parallel direct paths.* To solve serialization in the access to traffic, direct parallel paths between RSS queues and applications are required. This method, shown in Figure 3.3, achieves the best performance when a specific core is assigned both for taking packets from RSS queues and forwarding them to the user-level. This architecture also increases the scalability, because new parallel paths may be created on driver module insertion as the number of cores and RSS queues grows. In order to obtain parallel direct paths, we have to modify the data exchange mechanism between kernel-level and user-level.

In the downside, such technique mainly entails three drawbacks. First, it requires the use of several cores for capturing purposes, cores that otherwise may be used for other tasks. Second, packets may arrive potentially out-of-order at user-level, which may affect some kind of applications [WDC11]. Third, RSS distributes traffic to each receive queue by means of a hash function. When there is no interaction between flows, they can be analyzed independently, which allows taking the most of the parallelism by creating and linking one or several instances of a process to each capture core. As shown in Section 2.2, applications

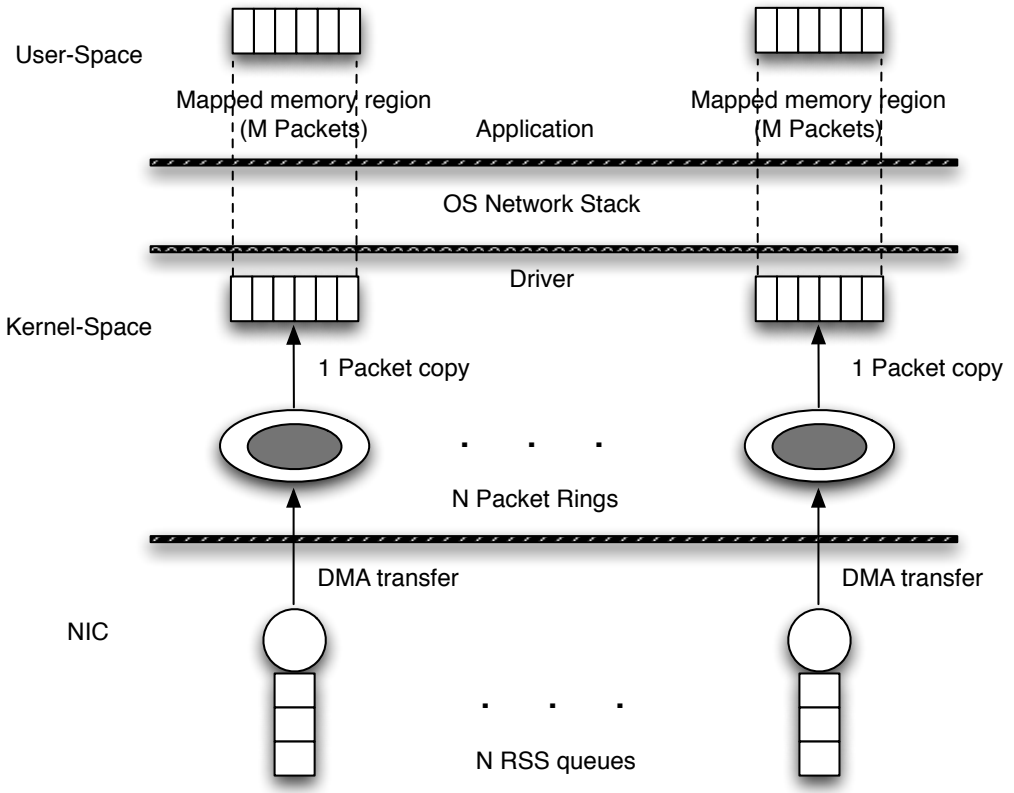


Figure 3.3: Optimized Linux Network Stack

such as software routers [HJPM10], and some NIDSs [VPI11, JLM+12], perfectly fit in this category. However, there are monitoring tasks that require analyzing different flows. One evident example is monitoring applications that inspect sessions instead of flows. A session is a couple of unidirectional flows that comprise the outgoing and incoming traffic, respectively, of a connection. As port numbers and IP addresses are inverted in these two flows, the current hash function developed in NICs forwards each of these flows to different queues. Another example is a VoIP monitoring system (as it will be shown in Chapter 6): assuming that such a system is based on the SIP protocol, it requires not only to monitor the signaling traffic (i.e., SIP packets) but also calls themselves—typically RTP traffic. SIP and RTP flows do not share

the 5-tuple that use the hash functions to assign packet to each queue, hence they are allocated to different queues and cores.

To face this, may be used two approaches, although both of them imply their own drawbacks. The simplest approach is that applications only receive traffic from a single core bound to a single RSS queue throughout its execution. Hence, applications are able to monitor those flows that require some interaction between them because it reads all the packets. This approach wastes the multi-core capacity of the systems but still take advantage of the rest of driver optimizations. That is, with respect to the traditional per-packet process, the system is able to capture and forward the traffic at high-rates to be processed at user space. However, the downside is that applications must be developed carefully to be fast enough to deal with these traffic rates, because parallelism in user-level is lost. The second option is that the capture system performs by itself some aggregation task. The idea is that each RSS queue is linked to a specific core to increase the CPU affinity. But before that, some block of the capture system must aggregate the traffic according to a given metric, and forward the traffic to user space—for example, to a socket queue. Then, applications must read from these socket queues, which aggregate the traffic that a given application expects. That is, this traffic includes all those flows that require interaction between them. For instance, if the application were tracking sessions, it would read both incoming and outgoing flows from the same socket. This is relatively simple for some tasks, for example the mentioned generation of sessions by applying symmetric (in terms of 5-tuple) hash functions [WP12]. However other tasks require more sophisticated modifications. Consider a VoIP monitoring system, as we have previously described. It requires to aggregate SIP and RTP flows of a given call, which use potentially different IP addresses and port number. This would require capture system to inspect SIP packets to identify their associated RTP flows, which is a challenge at high-speed rates. In any case, both approaches circumvent this problem at the expense of performance.

(iii) *Memory mapping.* Using this method, a standard application can map kernel memory regions, reading and writing them without intermediate copies. In this manner, we may map the DMA-able memory region where the NIC directly accesses. In such case, this technique is called zero-copy. As an inconvenient, exposing NIC rings and registers may entail risks for the stability of the system [Riz12a]. However, this is considered a minor issue as typically the provided Application Programming Interface (API)s protect NIC from incorrect accesses. In fact, all video boards use equivalent memory mapping techniques without major concerns. Another alternative is mapping the kernel packet memory region where driver copies packets from RX rings, to user-level, thus user applications access to packets without this additional copy. Such alternative removes one out of two copies in the default network stack. This technique is implemented on current GNU Linux as a standard raw socket with RX_RING/TX_RING socket option. Applying this method requires either driver-level or kernel-level modifications and in the data exchange mechanism between kernel-level and user-level.

(iv) *Batch processing.* To gain performance and avoid the degradation related with per-packet copies, batch packet processing may be applied. This solution groups packets into a buffer and copies them to kernel/user memory in groups called batches. Applying this technique permits to reduce the number of system calls and the consequent context switchings, and mitigates the number of copies. Thus, the overhead of processing and copying packets individually is removed. According to NAPI architecture, there are intuitively two points to use batches, first if packets are being asked in a polling policy, the engines may ask for more than one packet per request. Alternatively, if the packet fetcher works on an interrupt-driven basis, one intermediate buffer may serve to collect traffic until applications ask for it. The major problem of batching techniques is the increase of latency and jitter, and timestamp inaccuracy on received packets because packets have to wait until a batch is full or a timer expires [MSdRR+12], as it will be thoroughly

analyzed in Chapter 4. In order to implement batch processing, we must modify the data exchange between kernel-level and user-level.

(v) *Affinity and prefetching.* To increase performance and exploit memory locality, a process must allocate memory in a chunk assigned to the processor in which it is executing. This technique is called memory affinity. Other software considerations are CPU and interrupt affinities. CPU affinity is a technique that allows fixing the execution localization in terms of processors and cores of a given process (process affinity) or thread (thread affinity). The former action may be performed using Linux `taskset`[1] utility, and the latter by means of `pthread_setaffinity_np`[2] function of the Portable Operating System Interface (POSIX) pthread library. At kernel and driver levels, software and hardware interrupts can be handled by specific cores or processors using this same approach. This is known as interrupt affinity and may be accomplished writing a binary mask to `/proc/irq/IRQ#/smp_affinity`. The importance of setting capture threads and interrupts to the same core lies in the exploitation of cache data and load distribution across cores. Whenever a thread wants to access to the received data, it is more likely to find them in a local cache if previously these data have been received by an interrupt handler assigned to the same core. This feature in combination with the previously commented memory locality optimizes data reception, making the most of the available resources of a system. Another affinity issue that must be taken into account is to map the capture threads to the NUMA node attached to the PCIe slot where the NIC has been plugged. To accomplish such task, the system information provided by the `sysctl` interface (shown in Section 2.1.3) may result useful.

Additionally, in order to eliminate the inherent cache misses, the driver may prefetch the next packet (both packet data and packet descriptor) while the current packet is being processed. The idea behind prefetch-

[1] `linux.die.net/man/1/taskset`
[2] `linux.die.net/man/3/pthread_setaffinity_np`

ing is to load the memory locations that will be potentially used in a near future in processor's cache in order to access them faster when required. Some drivers, such as Intel `ixgbe`, apply several prefetching strategies to improve performance. Thus, any capture engine making use of such vanilla driver will see its performance benefited from the use of prefetching. Further studies such as [HJPM10, SZTG12] have shown that more aggressive prefetching and caching strategies may boost network throughput performance.

3.4 Novel Packet I/O Engines

In what follows, we present five capture engine proposals, namely: PF_RING DNA [RDC12], PacketShader [HJPM10], Netmap [Riz12a], PFQ [BDPGP12] and HPCAP [MSdRR+12], which have achieved significant performance. For each engine, we describe the system architecture (remarking differences with the other proposals), the abovementioned techniques that applies, what API is provided for clients to develop applications, and what additional functionality it offers. Table 3.1 shows a summary of the comparison of the proposals under study. We do not include some capture engines, previously proposed in the literature, because they are obsolete or unable to be installed in current kernel versions (Routebricks [DEA+09], UIO-IXGBE [Kra09]) or there is a new version of such proposals (PF_RING TNAPI [FD10]). Nevertheless, we start describing the fundamentals of one of them, Routebricks [DEA+09], as a precursor of this kind of systems.

3.4.1 Routebricks/Click

Although Routebricks may be obsolete, it is worthy to be explained because was one of the precursors of packet capturing using OTS systems. Routebricks [DEA+09] is an hybrid capturing/processing engine based on Click [KMC+00] modular router architecture. Click is a modular software for building routers. It is based on the concept of elements interconnected via a graph. Each element receives packets from an input port, processes

Table 3.1: Qualitative comparison of the five proposed capture engines (D=Driver, K=Kernel, K-U=Kernel-User interface)

Features/ Techniques	**PF_RING DNA**	**PS**	**netmap**	**PFQ**	**HPCAP**
Memory Pre-alloc. and re-use	✓	✓	✓	×/✓	✓
Parallel direct paths	✓	✓	✓	✓	✓
Memory mapping	✓	✓	✓	✓	✓
Zero-copy	✓	×	×	×	×
Batch processing	×	✓	✓	✓	×
CPU and interrupt affinity	✓	✓	✓	✓	✓
Memory affinity	✓	✓	×	✓	✓
Aggressive Prefetching	×	✓	×	×	✓
Level modifications	D,K, K-U	D, K-U	D,K, K-U	D (minimal), K,K-U	D, K-U
API	Libpcap-like	Custom	Standard libc	Socket-like	Custom

the packets and forwards them to the next interconnected element using an output port. Packets may be discarded or routed inside a processing element. Click code is located at kernel level and it is attached to Linux packet receive/send queues. Routebricks adds some new functionalities at both packet capturing and Click processing points. This capture engine modifies standard Intel 1/10 Gb/s drivers adding some new features to improve packet capture performance. The main ideas behind Routebricks driver modification are multi-queue support and batch processing.

Routebricks is able to reach 18.96 Mpps forwarding 64-byte packets, using four interfaces. This gives an approximate rate of 4.74 Mpps per interface. In spite of improving the performance of the standard driver and the operating system networking stack, Routebricks is not able to capture the maximum packet rate—14.88 Mpps per interface. The hardware setup used for performance evaluation experiments consists of one server with two Intel i7 (with Nehalem architecture [VM11]) processors each with four cores running at 2.8 GHz. Each processor has a memory module, following a NUMA architecture. Regarding connectivity, the server is equipped with two dual-port Intel 10Gb/s NICs.

One hint given by this capture engine is the fact that data placement on multi-processor system is not relevant at high-speed capture environments. This conclusion could be only applied taking into account capture rates obtained by this capture engine. As shown in [HJPM10] and previously explained in this chapter, increasing parallelism in memory access speeds up the performance of packet capture at high speed environments. Routebricks system may be easily build in a general-purpose server following the detailed instructions in its webpage [ACE09] . The release includes a Linux kernel configuration file, a patched version of Intel 10 Gb/s driver for Linux and the Click elements needed for using multiple reception/transmission queues.

3.4.2 PF_RING DNA

PF_RING Direct NIC Access (DNA) is a framework and engine to capture packets based on Intel 1/10 Gb/s cards. This engine implements pre-

allocation and re-use of memory in all its processes, both RX and PF_RING queue allocations. PF_RING DNA also allows building parallel paths from hardware receive queues to user processes, that is, it allows to assign a CPU core to each received queue whose memory can be allocated observing NUMA nodes, permitting the exploitation of memory affinity techniques.

Differently from the other proposals, it implements full zero-copy, that is, PF_RING DNA maps user-space memory into the DMA-able memory region of the driver allowing users' applications to directly access to card registers and data in a DNA fashion. In such a way, it avoids the intermediation of the kernel packet buffer reducing the number of copies. However, as previously noted, this is at the expense of a slight weakness to errors from users' applications that occasionally do not follow the PF_RING DNA API (which explicitly does not allow incorrect memory accesses), which may potentially imply system crashes. In the rest of the proposals, direct accesses to the NIC are protected. PF_RING DNA behavior is shown in Figure 3.4, where some NAPI steps have been replaced by a zero-copy technique.

PF_RING DNA API provides a set of functions for opening devices to capture packets. It works as follows: first, the application must be registered with `pfring_set_application_name()` and before receiving packets, the reception socket can be configured with several functions, such as, `pfring_set_direction()`, `pfring_set_socket_mode()` or `pfring_set_poll_duration()`. Once the socket is configured, it is enabled for reception with `pfring_enable_ring()`. After the initialization process, each time a user wants to receive data `pfring_recv()` function is called. Finally, when the user finishes capturing traffic `pfring_shutdown()` and `pfring_close()` functions are called. This process is replicated for each receive queue.

As one of the major advantages of this solution, PF_RING API comes with a wrapping to the abovementioned functions, which provides large flexibility and ease of use, essentially following the de facto standard of the libpcap library. Additionally, the API provides functions for applying filtering rules, e.g., Berkeley Packet Filter (BPF) filters, network bridging, and IP reassembly. PF_RING DNA and a user library for packet processing are free-available for the research community [DC12].

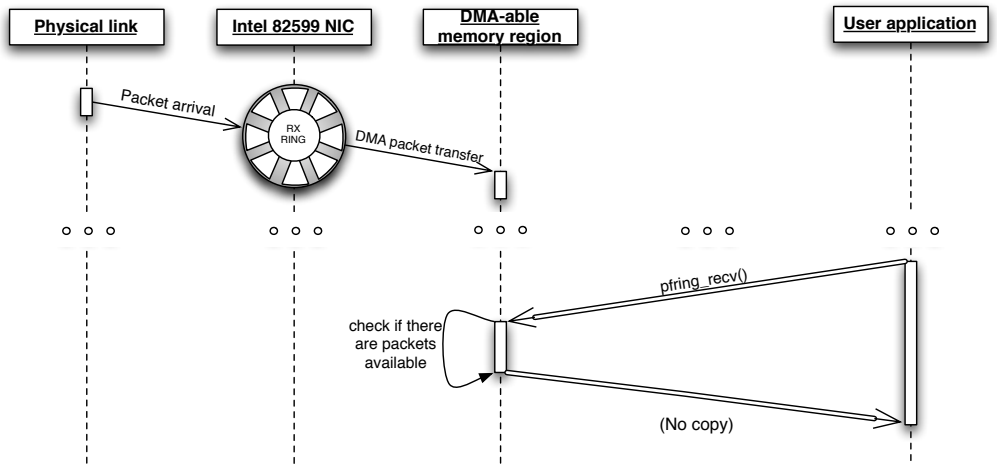


Figure 3.4: PF_RING DNA RX scheme

3.4.3 PacketShader

The authors of PacketShader (PS) developed their own capture engine to highly optimize the traffic capture module as a first step in the process of developing a software router able to work at multi-10Gb/s rates. However, all their efforts are applicable to any generic task that involves capturing and processing packets. They apply memory pre-allocation and re-use, specifically, two memory regions are allocated—one for the packet data, and another for its metadata. Each buffer has fixed-size cells corresponding to one packet. The size for each cell of packet data is aligned to 2048 bytes, which corresponds to the next highest power of two for the standard Ethernet Maximum Transmission Unit (MTU). Metadata structures are compacted from 208 bytes to only 8 bytes (96%) removing unnecessary fields for many networking tasks.

Additionally, PS implements memory mapping, thus allowing users to access to the local kernel packet buffers avoiding unnecessary copies. In this regard, the authors highlight the importance of NUMA-aware data placement in the performance of its engine. Similarly, it provides parallelism to packet processing at user-level, which balances CPU load and gives scalability in

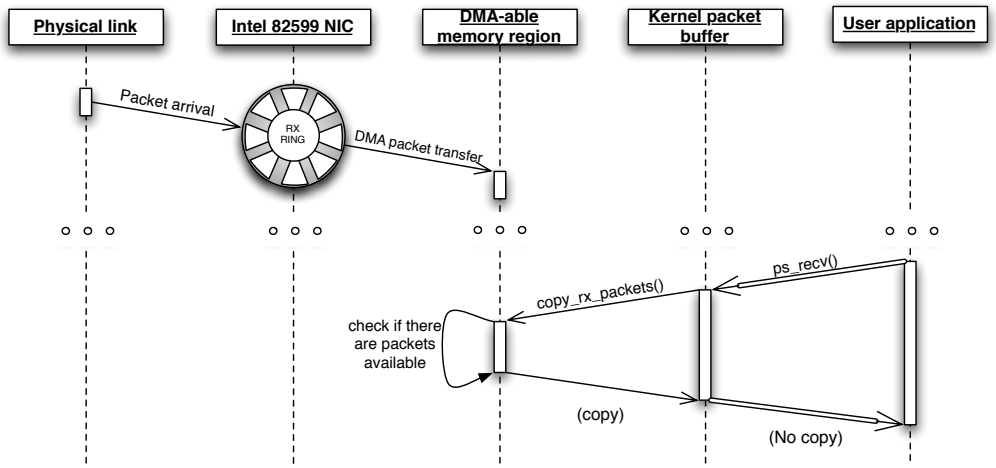


Figure 3.5: PacketShader RX scheme

the number of cores and queues.

To reduce the per-packet processing overhead, batching techniques are utilized in user-level. For each batch request, the driver copies data from the huge packet buffer to a consecutive mapped memory region, which is accessed from user-level. In order to eliminate the inherent cache misses, the modified device driver prefetches the next packet (both packet data and packet descriptor) while the current packet is being processed.

PS API works as follows: (i) user application opens a char device to communicate with the driver, `ps_init_handle()`, (ii) attaches to a given reception device (queue) with an `ioctl()`, `ps_attach_rx_device()`, and (iii) allocates and maps a memory region, between the kernel and user levels to exchange data with the driver, `ps_alloc_chunk()`. Then, when a user application requests for packets by means of an `ioctl()`, `ps_recv_chunk()`, PS driver copies a batch of them, if available, to the kernel packet buffer. PS interaction with users' applications during the reception process is summarized in Figure 3.5.

PS I/O engine is available for the community [HJPM12]. Along with the modified Linux driver for Intel 82598/82599-based NICs network interface cards, a user library is released in order to ease the usage of the driver.

The release also includes several sample applications, namely: a simplified version of `tcpdump` [PCA10], an `echo` application which sends back all traffic received by one interface, and a packet generator which is able to generate UDP packets with different 5-tuple combinations at maximum speed.

3.4.4 Netmap

Netmap proposal shares most of the characteristics of PacketShader's architecture. That is, it applies memory pre-allocation during the initialization phase, buffers of fixed sizes (also of 2048 bytes), batch processing and parallel direct paths. It also implements memory mapping techniques to allow users' application to access to kernel packet buffers (direct access to NIC is protected) with a simple and optimized metadata representation.

Such simple metadata is named netmap memory ring (Figure 3.6), and its structure contains information such as the ring size, a pointer to the current position of the buffer (*cur*), the number of received packets in the buffer or the number of empty slots in the buffer, in reception and transmission buffers respectively (*avail*), flags about the status, the memory offset of the packet buffer, and the array of metadata information; it has also one slot per packet which includes the length of the packet, the index in the packet buffer and some flags. Note that there is one netmap ring for each RSS queue, reception and transmission, which allows implementing parallel direct paths.

Netmap API usage is intuitive: first, a user process opens a netmap device with an `ioctl()`. To receive packets, users ask the system the number of available packets with another `ioctl()`, and then, the lengths and payloads of the packets are available for reading in the slots of the netmap_ring. This reading mode is able to process multiple packets in each operation. Note that netmap supports blocking mode through standard system calls, such as `poll()` or `select()`, passing the corresponding netmap file descriptors. In addition to this, netmap comes with a library that maps libpcap functions into own netmap ones, which facilitates its operation. As a distinguish characteristic, Netmap works in an extensive set of hardware solutions: Intel 10 Gb/s adapters and several 1Gb/s adapters—Intel, RealTek and nVidia.

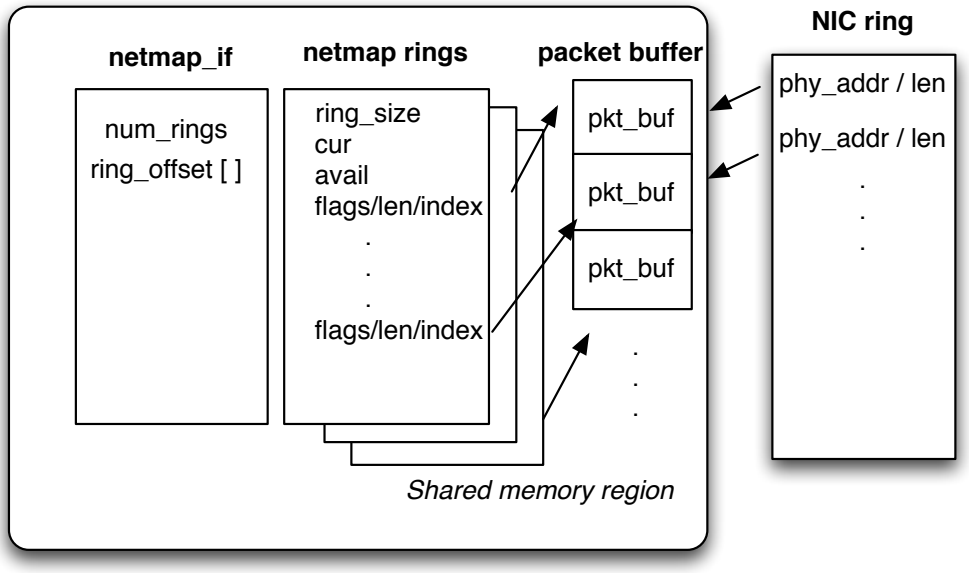


Figure 3.6: Netmap data structure

Netmap presents other additional functionalities as, for example, packet forwarding.

Netmap framework is available for FreeBSD (HEAD, stable/9 and stable/8) and for Linux [Riz12b]. The current netmap version consists of 2000 lines for driver modifications and system calls, as well as a C header file of 200 lines to help developers to use netmap's framework from user applications.

3.4.5 PFQ

PFQ is a novel packet capture engine that allows packet sniffing in user applications with a tunable degree of parallelism. The approach of PFQ is different from the previous studies. Instead of carrying out major modifications to the driver in order to skip the interrupt scheme of NAPI or map DMA-able memory and kernel packet buffers to user-space, PFQ is a general architecture that allows using both modified and vanilla drivers.

PFQ follows NAPI to fetch packets but implements two novel modifications once packets arrive at the kernel packet buffer with respect to the

standard networking stack. First, PFQ uses an additional buffer (referred as batching queue) in which packets are copied once the kernel packet buffer is full, those packets are copied in a single batch that reduces concurrency and increases memory locality. This modification may be classified both as a batching and memory affinity technique. As a second modification, PFQ makes the most of the parallel paths technique at kernel level, that is, all its functionalities execute in parallel and in a distributed fashion across the system's cores, which has proven to minimize the overhead. In fact, PFQ is able to implement a new layer, named Packet Steering Block, in between user-level and batching queues, providing some interesting functionalities. Such layer distributes the traffic across different receive sockets (without limitation on the number of queues than can receive a given packet). These distribution tasks are carried out by means of memory mapping techniques to avoid additional copies between such sockets and the user level. The Packet Steering Block allows a capture thread to move a packet into several sockets, thus a socket may receive traffic from different capture threads. This functionality circumvents one of the drawbacks of using the parallel paths technique, that is, scenarios where packets of different flows or sessions must be analyzed by different applications—as explained in Section 3.3. Figure 3.7 shows a temporal scheme of the process of requesting a packet in this engine.

It is worth remarking that, as stated before, PFQ obtains good performance with vanilla drivers, but using a patched driver with minimal modifications (a dozen lines of code) improves such performance. The driver change is to implement memory pre-allocation and re-use techniques.

PFQ is an open-source package that consists of a Linux kernel module and a user-level library written in C++, available under General Public License (GPL) license [BDPP13]. PFQ API defines a `pfq` class, which contains methods for initialization and packet reception. Whenever a user wants to capture traffic: (i) a `pfq` object must be created using the provided C++ constructor, (ii) devices must be added to the object calling its `add_device()` method, (iii) timestamping must be enabled using `toggle_time_stamp()` method, and (iv) capturing must be enable using `enable()` method. After the initialization, each time a user wants to read a group of packets, the `read()` method

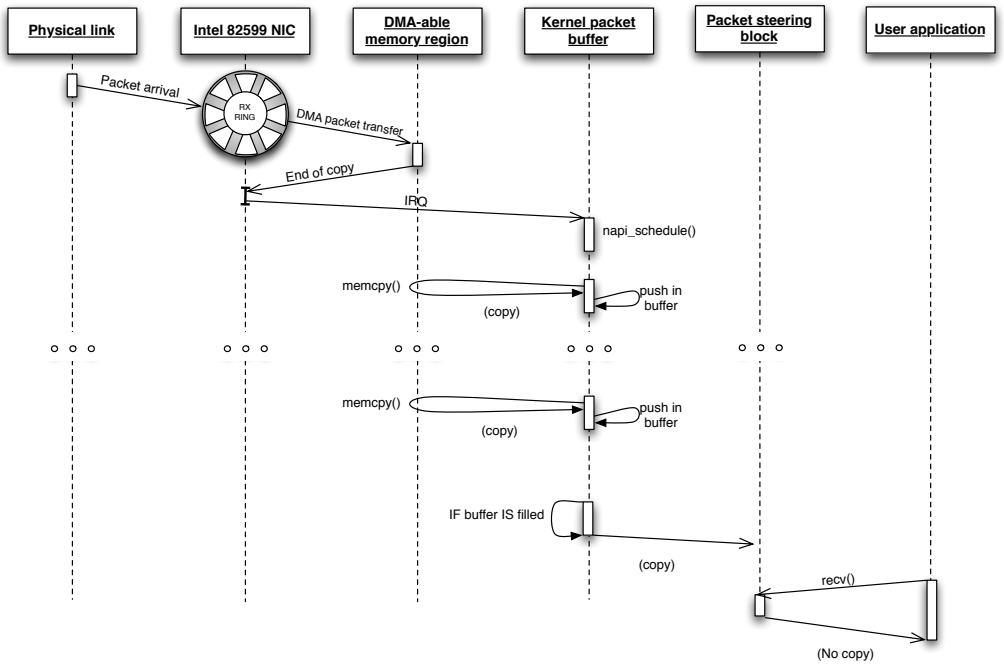


Figure 3.7: PFQ RX scheme

is called. Using a custom C++ iterator provides by PFQ, the user can read each packet of the received group. When a user-level application finishes `pfq` object is destroyed by means of its defined C++ destructor. To get statistics about the received traffic `stats()` method can be called.

3.4.6 HPCAP

High-performance Packet CAPture (HPCAP) is a packet sniffing solution based on similar principles as previous engines, but with some significant differences, namely: (i) accurate packet timestamping, and (ii) overlapping capture and different processing tasks.

HPCAP is implemented on a kernel-level thread—one per each NIC's receiving queue. Such thread is constantly polling its corresponding receive descriptor ring for new incoming packets. If a new packet is detected, the sniffing thread copies the packet to its corresponding packet buffer. Figure 3.8

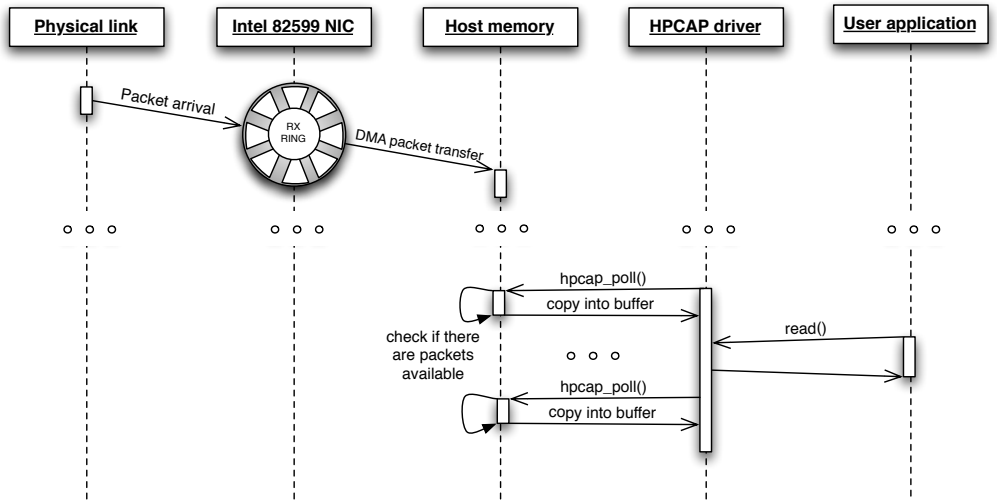


Figure 3.8: HPCAP RX scheme

illustrates such process.

Packets are timestamped before being copied and, consequently, timestamping is more accurate than with other capture engines. This improvement on timestamping accuracy is at expense of CPU load. Packet timestamping process will be thoroughly analyzed in Chapter 4.

The packet buffer is circular and its memory is previously allocated (before starting to capture) avoiding per-packet dynamic allocations. Its length may be configured and its default value is 1 GB. Due to kernel memory limitations, all threads of the modules must share such 1 GB of memory. For instance, if we have one interface with 4 queues, the buffer length of each queue is 256 MB. Each position of the buffer contains a packet header including packet timestamp (second and nanosecond), capture length (packets may be capped) and actual length. The packet buffer is also mapped. Thus, user-level threads and processes are able to read in this memory region without intermediate copies.

The system supports multiple listeners (threads or process) to fetch packets from the same packet buffer in a single-producer/multiple-consumer basis. That is, the packet buffer has one write pointer (where traffic sniffer copies

packets) and multiple read pointers (where each listener reads packets). In order to keep data consistency, the packet read throughput would be set by the slowest listener. In order to achieve peak performance both traffic sniffer thread and the multiple listeners must be executed in the same NUMA node [HJPM10]. Additionally, such NUMA node must be the assigned to the PCIe slot where the NIC has been connected. This distinguishing feature allows monitoring application to focus on packet processing, while, for example, a different application stores them into non-volatile volumes, thus overlapping data storing and processing and exploiting parallelism, which entails better performance.

HPCAP API workflow is similar as previous explained engines, such as PacketShader or netmap: user application opens a char device to communicate with the driver and attaches to a given reception queue with an `ioctl()`, `hpcap init_open()`. Then, HPCAP allows two reading mode. On the one hand, user application may directly access to the packet buffer by means of the standard `read()` function (e.g., we may run a `dd` process to dump packet buffer into disk). On the other hand, a user application may map the corresponding packet buffer to exchange data with the driver, `hpcap_map()` and access to the packet kernel packet buffer by means of an `ioctl()`: `hpcap_wait_timeout()`. With this function, we obtain the number of available bytes in the buffer and acknowledge an amount of bytes already processed. Note that this ACK mechanism is necessary to coordinate the single polling thread with the multiple listeners. We can configure blocking or non-blocking (with timeout expiration) wait.

3.5 Testbed

Once we have detailed the main characteristics of the most prominent capture engines in the literature, we turned our focus to their performance in terms of percentage of packets correctly received. It is noteworthy that the comparison between them, from a quantitative standpoint, is not an easy task for two reasons: first, the hardware used by the different studies is not equivalent (in terms of type and frequency of CPU, amount and frequency of main memory,

server architecture and number of network cards); second, the performance metrics used in the different studies are not the same, with differences in the type of traffic, and measurement of the burden of CPU or memory. For such reason, we have stress tested the engines described in the previous section, on the same architecture.

3.5.1 Hardware and Software Setup

Our testbed setup consists of two machines (one for traffic generation purposes and another for receiving traffic and evaluation) directly connected through a 10 Gb/s fiber-based link. The receiver side is based on Intel Xeon with two processor of 6 cores each running at 2.30 GHz, with 96 GB of Double Data Rate type 3 (DDR3) Synchronous Dynamic Random Access Memory (SDRAM) at 1333 MHz and equipped with a 10 GbE Intel NIC based on 82599 chip. The server motherboard model is Supermicro X9DR3-F with two processor sockets and three PCIe 3.0 slots per processor, directly connected to each processor, following a similar scheme to that depicted in Figure 2.2(b). The NIC is connected to a slot corresponding to the first processor. The sender uses a HitechGlobal HTG-V5TXT-PCIe card which contains a Xilinx Virtex-5 FPGA (XC5VTX240) and four 10 GbE Small Form-factor Pluggable (SFP)+ ports. Using such a hardware-based sender guarantees accurate packet interarrivals and 10 Gb/s throughput regardless of packet sizes.

On software side, the receiver has installed a 12.04 Ubuntu with Linux kernel 3.2.16. Now, let us describe the commands and applications used to configure the driver and receive traffic, for each capture engine. In the case of PF_RING, we installed driver using the provided script `load_dna_driver.sh`, changing the number of receive queues with the RSS parameter in the insertion of the driver module. To receive traffic using multiple queues, we executed the following command: `pfcount_multichannel -i dna0 -a -e 1 -g 0:1:...:n`, where `-i` indicates the device name, `-a` enables active waiting, `-e` sets reception mode and `-g` specifies the thread affinity for the different queues. Regarding PS, we installed the driver using the provided

script `install.py` and receive packets using a slightly modified version of the provided application `echo`. Finally, in the case of PFQ, we installed the driver using n reception queues, configured the receive interface, eth0, and set the Interrupt ReQuest (IRQ) affinity with the followings commands:`insmod ./ixgbe.ko RSS=n,n; ethtool -A eth0 autoneg off rx off tx off; bash ./set_irq_affinity.sh eth0`. To receive packets from eth0 using `n` queues with the right CPU affinity, we ran: `./pfq-n-counters eth0:0:0 eth0:1:1 ... eth0:n:n`.

Note that in all cases, we have paid attention to NUMA affinity by executing the capture threads in the processor that the NIC is connected, as it has 6 cores, this is only possible when there are less than seven concurrent threads. In fact, ignoring NUMA affinity implies extremely significant performance cuts, specifically in the case of the smallest packet sizes, this may reduce performance by its half. Regarding PFQ, we evaluated its performance installing the aware driver.

3.5.2 Test Traffic Dataset

To stress test the different capture engines, we have used both synthetic traffic and real traces. On the one hand, synthetic traffic is composed by fix-sized TCP packets with incremental IP addresses and port numbers. The packet payload is randomly generated.

On the other hand, we use a real trace, which was sniffed at an OC192 (9953 Mb/s) backbone link of a Tier-1 ISP located between San Jose and Los Angeles (both directions), available from CAIDA [WAcA09]. The average packet size in the trace is 743 Bytes.

3.6 Performance Evaluation

In this section, we discuss the performance evaluation results, highlight the advantages and drawbacks of each capture engine and give guidelines to the research community in order to choose the more suitable capture system.

It is worth remarking that netmap does not appear in our comparison

because its Linux version does not allow changing the number of receive queues being this fixed at the number of cores. As our testbed machine has 12 cores, in this scenario netmap capture engine requires allocating memory over the kernel limits, and netmap does not start. However, we note that according to [Riz12a], its performance figures should be similar to those from PacketShader.

3.6.1 A Worst-Case Scenario

In this first experiment, we aim to analyze the worst case scenario, i.e., a full-saturated 10 Gb/s link with packets of constant minimum size of 60 bytes (as in the following, excluding Ethernet Cyclic Redundancy Check (CRC)) for different number of queues—ranging from 1 to 12. Note that this represents an extremely demanding scenario, 14.88 Mpps, but probably not very realistic given that the average Internet packet size is clearly larger [CAI12]. Results are shown in Figure 3.9.

In this scenario, PacketShader is able to handle nearly the total throughput when the number of queues ranges between 1 and 4, being with this latter figure when the performance peaks. Such relatively counterintuitive behavior is shared by PF_RING DNA system, which shows its best permanence, a remarkable 100% packet received rate, with a few queues, whereas with when number of queues is larger than 7, the performance dips. Conversely to such behavior, PFQ increases its performance according to the number of queues up to its maximum with nine queues, when such growth stalls. In the case of HPCAP, the performance ranges between 75 and 90% out of the total packets sent, achieving its maximum throughput with two queues. Note that HPCAP is only able to use 6 queues in our testbed of 12 cores, because two threads per RSS queue are necessary: one for polling in kernel-level and one for sniffing in user-level.

3.6.2 Scalability Analysis

To further investigate the scalability phenomenon for each capture engine, Figures 3.10, 3.11, 3.12 and 3.13 depict the results for PF_RING DNA, Pack-

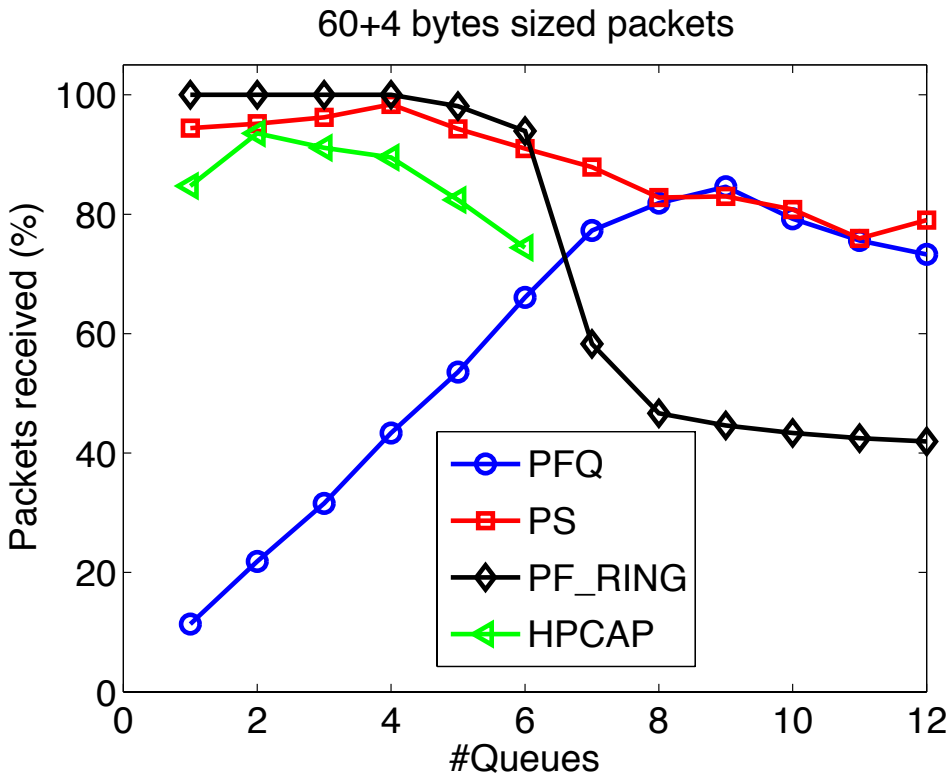


Figure 3.9: Engines' performance for 60 (+4 CRC) byte packets

etShader, PFQ and HPCAP respectively, for different packets sizes (60, 64, 128, 256, 512, 1024, 1250 and 1500 bytes) and one, six and twelve queues. PF_RING DNA shows the best results with one and six queues. It does not show packet losses for all scenarios but those with packet sizes of 64 bytes and, even in this case, such figure is very low (about 4% with six queues and lower than 0.5% with one). Surprisingly, increasing packet sizes from 60 to 64 bytes entails degradation of the PF_RING DNA performance, although beyond this packet size, the performance recovers 100% rates. Note that larger packet sizes directly imply lower throughputs in Mpps. According to [Riz12a], investigation in this regard has shown that this behavior is because of the design of NICs and I/O bridges that make certain packet sizes to fit better with their architectures.

In a scenario in which one single user-level application is unable to handle

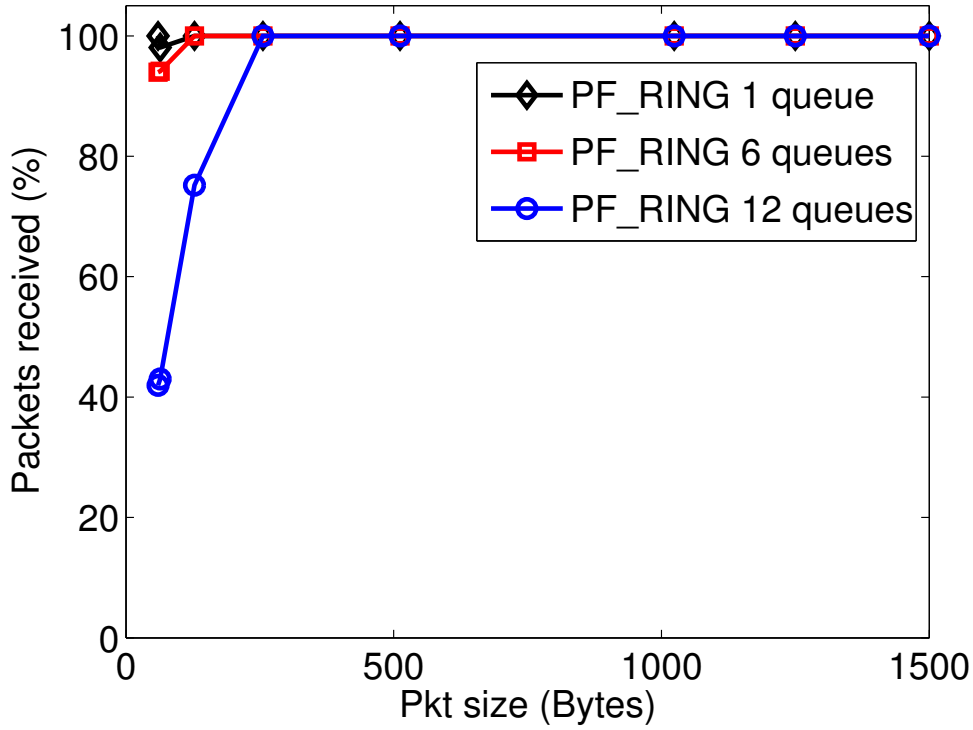


Figure 3.10: PF_RING DNA performance in terms of percentage of received packet for different number of queues and constant packet sizes for a full-saturated 10 Gb/s link

all the received traffic, may result of interest to use more than one receive queue (with one user-level application per queue). In our testbed and assuming twelve queues, PacketShader has shown comparatively the best result, although, as PF_RING DNA, it performs better with a fewer number of queues. Specifically, for packet sizes of 128 bytes and larger ones, it achieves full packet received rates, regardless the number of queues. With the smallest packets sizes, it gives loss ratios of 20% in its worst case of twelve queues, 7% with six, and about 4% with one queue.

Analyzing PFQ's results, we note that such engine achieves also 100% received packet rates, but conversely to the other approaches, it works better with several queues. It requires at least six ones to achieve no losses with

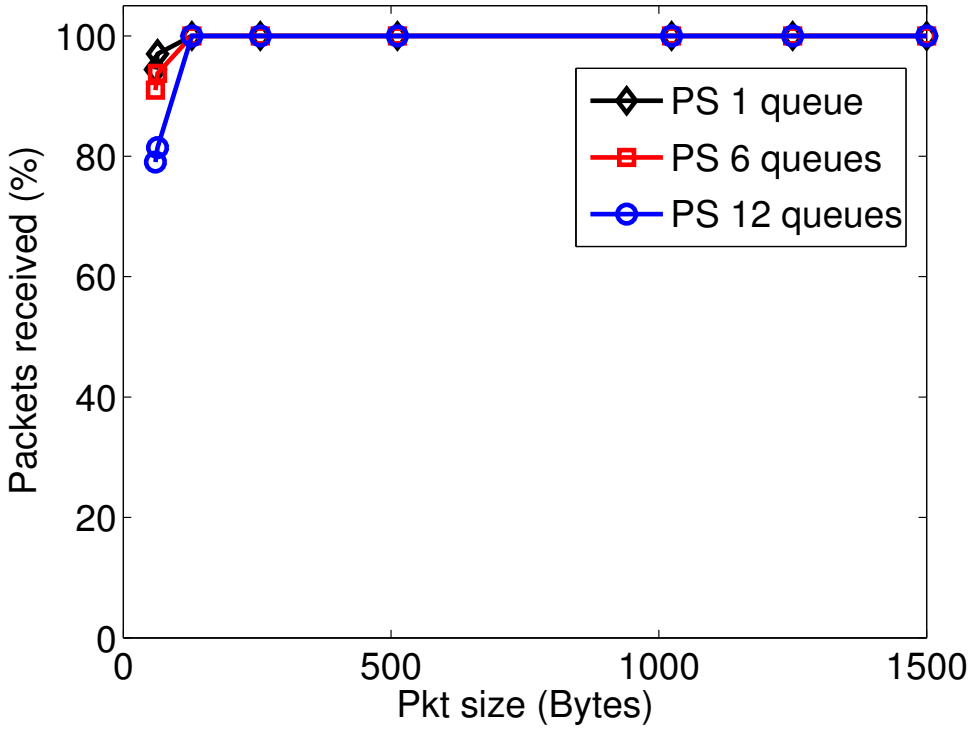


Figure 3.11: PacketShader performance in terms of percentage of received packet for different number of queues and constant packet sizes for a full-saturated 10 Gb/s link

packets of 128 bytes or more, whereas with one queue, packets must be larger or equal to 1000 bytes to achieve full rates. This behavior was expected due to the importance of parallelism in the implementation of PFQ.

HPCAP presents a similar behavior to PacketShader and PF_RING. That is, it achieves the maximum throughput (zero losses) with packet size greater than 64 bytes, although the performance is below with minimum packet size. We tested with 1, 3 an 6 queues because, as previously explained, we cannot configure more than 6 out of 12 queues.

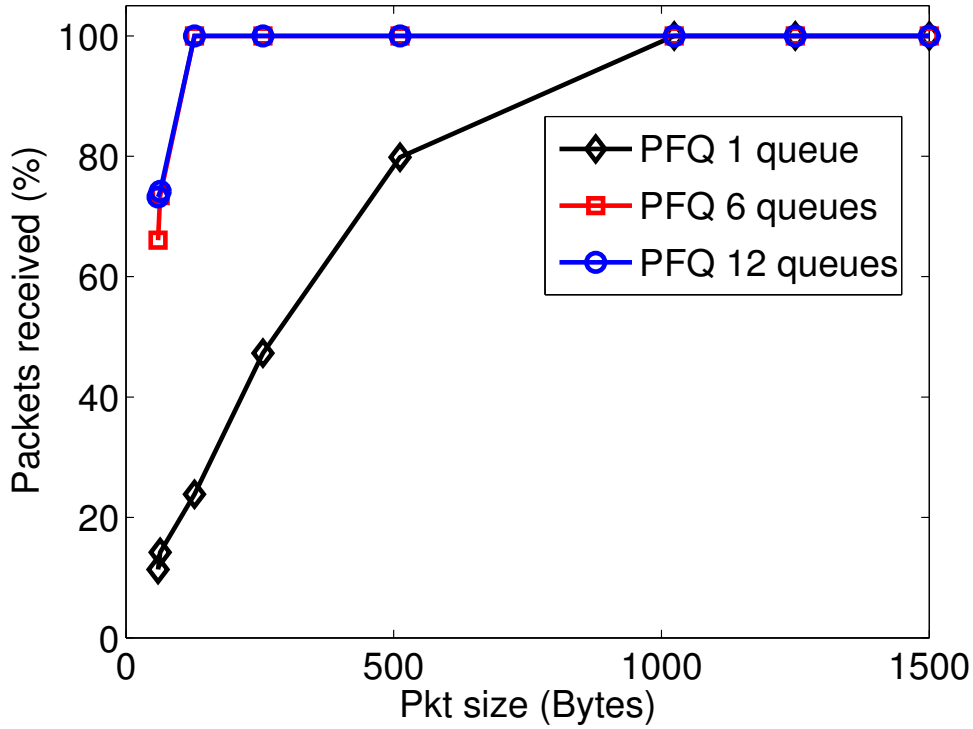


Figure 3.12: PFQ performance in terms of percentage of received packet for different number of queues and constant packet sizes for a full-saturated 10 Gb/s link

3.6.3 A Stressful Real Scenario

In this section, we assess each capture engine sending a real trace (described in Section 3.5) at maximum speed (10 Gb/s) for different number of queues—ranging from 1 to 12. Although this scenario is not realistic in terms of bitrate (is unlikely to full-saturate a backbone link) but in this experiment the packet size follows a real (empirical) distribution.

Figure 3.14 shows the obtained results. All capture engines in all configurations (number of queues), except PFQ configured with one queue, are able to receive 100% of packets without packet losses. This shows that the different proposals are able to sniff traffic in a real scenario at line-rate.

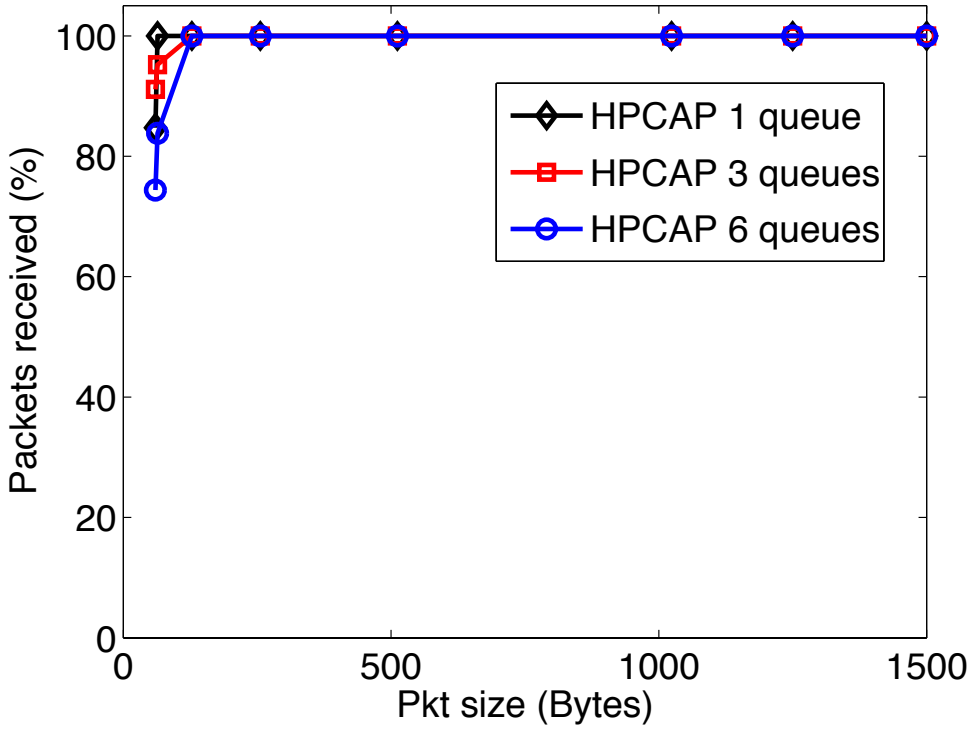


Figure 3.13: HPCAP performance in terms of percentage of received packet for different number of queues and constant packet sizes for a full-saturated 10 Gb/s link

3.6.4 Findings and Guidelines

We find that these engines may cover different scenarios, even the more demanding ones. We state two types of them, whether we may assume the availability of multiple cores or not, and whether the traffic intensity (in Mpps) is extremely high or not (for example, packet size averages smaller than 128 bytes, which is not very common). That is, if the number of queues is not relevant, given that the capture machine has many available cores or no other process is executing but the capture process itself, and the intensity is relatively low (namely, some 8 Mpps), PFQ seems to be a suitable option. It comprises a socket-like API, which is intuitive to use, as well as other interesting functionalities, such as an intermediate layer to aggregate traffic,

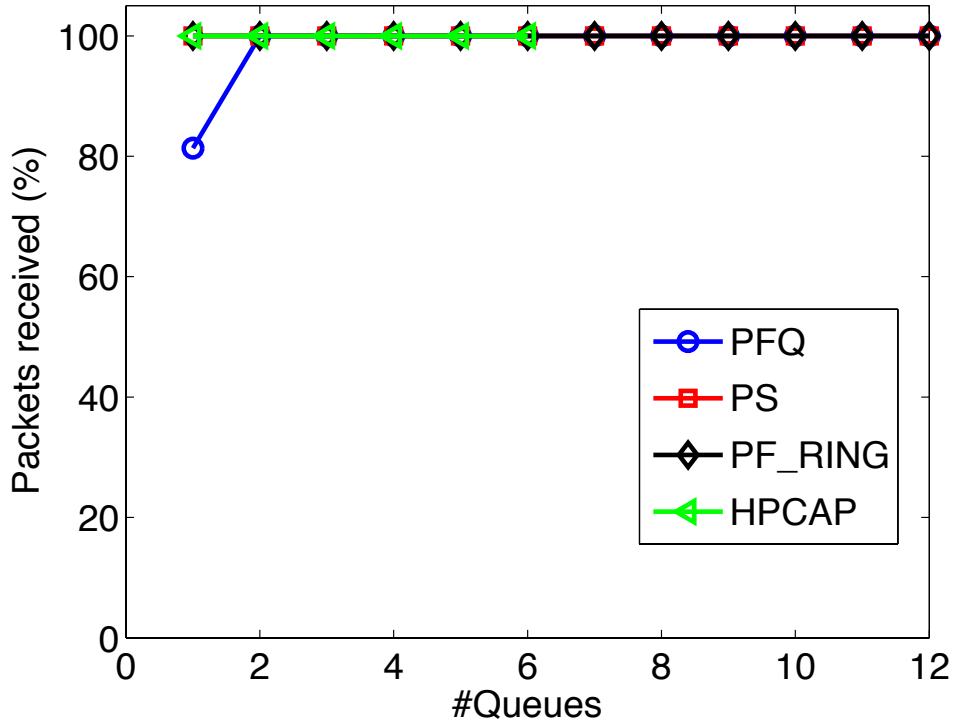


Figure 3.14: Engines' performance in a real scenario (CAIDA trace)

while it achieves full-received packet rates for twelve queues. On the other hand, if traffic intensity is higher than the previous assumption, PacketShader presents a good compromise between the number of queues and performance.

Nonetheless, often multi-queue scenarios are not adequate. For example, packet disorder may be a significant inconvenient (according to the application running on the top of the engine) [WDC11], or simply, it may be interesting to save cores for other tasks. In this scenario, PF_RING DNA and PacketShader are clearly the best options, as it shows (almost) full rates regardless packet sizes even with only one queue (thus, avoiding any drawbacks due to parallel paths).

Finally, HPCAP is shown as the best choice when accurate timestamps are necessary [MSdRR+12], achieving almost the same good results as PF_RING and PacketShader with only one queue, and flexibility is required (because

HPCAP allows us to process with multiple listeners and overlap capture and processing).

3.7 Summary and Conclusions

The utilization of commodity hardware in high-performance tasks, previously reserved to specialized hardware, has raised great expectation in the Internet community, given the astonishing results that some approaches have attained at low cost. In this chapter we have first identified the limitations of the default networking stack and shown the proposed solutions to circumvent such limitations. In general, the keys to achieve high performance are efficient memory management, low-level hardware interaction and programming optimization. Unfortunately, this has transformed network monitoring into a non-trivial process composed of a set of sub-tasks, each of which presents complicated configuration details. The adequate tuning of such configuration has proven of paramount importance given its strong impact on the overall performance. In this light, this chapter has carefully reviewed and highlighted such significant details, providing practitioners and researchers with a road map to implement high-performance networking systems in commodity hardware. Additionally, we note that this effort of reviewing limitation and bottlenecks and their respective solutions may be also useful for other areas of research and not only for monitoring purposes or packet processing (for example, virtualization).

This chapter has also reviewed and compared successful implementations of packet capture engines. We have identified the solutions that each engine implements as well as their pros and cons. Specifically, we have found that each engine may be more adequate for a different scenario according to the required throughput and availability of processing cores in the system. Specifically:

(i) PF_RING and Packetshader achieve wire-speed using a few cores (even with only one RSS queue) but they may lack of flexibility.

(ii) HPCAP obtains similar results, although performance is degraded with

the minimum packet length of 60 bytes) and one additional core is needed for sniffing. However, HPCAP provides accurate timestamping and greater parallelism capabilities for traffic processing.

(iii) PFQ needs a greater amount of queues and cores to achieve an acceptable throughput but gives more flexibility and additional functionalities, such as customized packet aggregation and a socket-like API.

As a conclusion, the performance results exhibited in this chapter, in addition to the inherent flexibility and low cost of the systems based on commodity hardware, make this solution a promising technology at the present. Finally, we highlight that the analysis and development of software based on multi-core hardware is still an open issue. Problems such as the aggregation of related flows, accurate packet timestamping, and packet disordering will for sure receive more attention by the research community in the future. Particularly, packet timestamping issues will be analyzed in Chapter 4 whereas flow matching will be studied in Chapter 5.

Chapter 4

Analysis of Timestamp Accuracy of High-Performance Packet I/O Engines

Along with captured packets is desirable to get accurate timestamps—i.e., the date and time of day when a packet was received. Software timestamping requires system calls to obtain the corresponding timestamps requesting to the system clock—whereas hardware timestamping is only possible using specialized solutions. Note that monitoring a 10 Gb/s link entails interarrival times less than 1 μs, and consequently, to achieve such accuracy and precision of timestamp is actually a challenging task. In this chapter, we discover and quantify the timestamping inaccuracy introduced by novel high-performance packet I/O engines. Hence, we propose two techniques to overcome or mitigate such issue.

4.1 Introduction

While the improvements explained in Chapter 3 boost up the performance of packet capture engines, surprisingly, packet timestamp capabilities have been shifted to the background, despite their importance in monitoring tasks. Typically, passive network monitoring requires not only capturing packets

but also labeling them with their arrival timestamps. In fact, the packet timestamp accuracy is relevant to the majority of monitoring applications but it is essential in those services that follow a temporal pattern. As an example, in a VoIP monitoring system, signaling must be tracked prior to the voice stream. Moreover, the use of packet batches as a mechanism to capture traffic causes the addition of a source of inaccuracy in the process of packet timestamping. Essentially, when a high-level layer asks for packets the driver stores and forwards them to the requestor at once. Therefore, all the packets of a given batch have nearby timestamps whereas inter-batch times are huge, not representing real interarrival times. This phenomenon has not received attention to date. Consequently, in this chapter we assess the timestamp accuracy of novel packet capture engines and propose two different approaches to mitigate the impact of batch processing.

The rest of the chapter is organized as follows: Section 4.2 describes the problem of timestamping when capturing with OTS systems, detailing the different sources of accuracy and precision degradation, mainly, caused by batch processing. In Section 4.3, we propose three techniques, which overcome or mitigate the timestamp issue previously described. In Section 4.4, we evaluate the timestamp accuracy of the proposed methods and give a comparison with previous solutions. Finally, Section 4.5 concludes the chapters highlighting the main findings.

4.2 Problem Statement: Timestamp Accuracy Degradation Sources

Dealing with high-speed networks claims for advanced timing mechanisms. For instance, at 10 Gb/s a 60-byte sized packet is transferred in 67.2 ns: $(60 + 4\ (\text{CRC}) + 8\ (\text{Preamble}) + 12\ (\text{Inter-Frame Gap})) \cdot 8 \cdot 10^{-10}$, whereas a 1514-byte packet in 1230.4 ns. In the light of such demanding figures, packet capture engines should implement timestamping policies as accurate as possible.

All capture engines suffer from timestamp inaccuracy due to kernel schedul-

ing policy because other higher priority processes make use of CPU resources. Such problem becomes more dramatic when batch timestamping is applied. In that case, although incoming packets are copied into kernel memory and timestamped in a 1-by-1 fashion, this copy-and-timestamp process is scheduled in time quanta whose length is proportional to the batch size. Thus, packets received within the same batch will have an equal or very similar timestamp. In Figure 4.1 this effect is exposed for a 100%-loaded 10 Gb/s link in which 60-byte packets are being received using PacketShader [HJPM10], i.e., a new packet arrives every 67.2 ns (black dashed line). As shown, packets received within the same batch do have very little interarrival time (corresponding to the copy-and-timestamp duration), whereas there is a huge interarrival time between packets from different batches. Therefore, the measured interarrival times are far from the real values.

Figure 4.2 shows the standard deviation of the observed timestamp error after receiving 1514-byte sized packets for one second at maximum rate. The timestamp accuracy is degraded with batch size. Note that when the batch size is beyond 16 packets, the error tends to stall because the effective batch size remains almost constant —although a given batch size is requested to the driver, user applications will be only provided with the minimum between the batch size and the number of available packets. We notice that PFQ [BDPGP12] does not use batch processing at driver-level and this source of inaccuracy does not affect its timestamping. However, timestamp inaccuracy may be added due to the per-packet processing latency.

At the same time, other sources of inaccuracy appear when using more than one hardware queue and trying to correlate the traffic dispatched by different queues. On the one hand, interarrival times may even be negative due to packet reordering, as shown in [WDC11]. On the other hand, the lack of low-level synchronism among different queues must be taken into account as different cores of the same machine cannot concurrently read the timestamp counter register [BRV09]. PFQ suffers from these effects because it must use multiple queues in order to achieve line-rate packet capture. However, batch-oriented drivers, such as PacketShader, are able to capture wire-speed traffic using just one hardware queue.

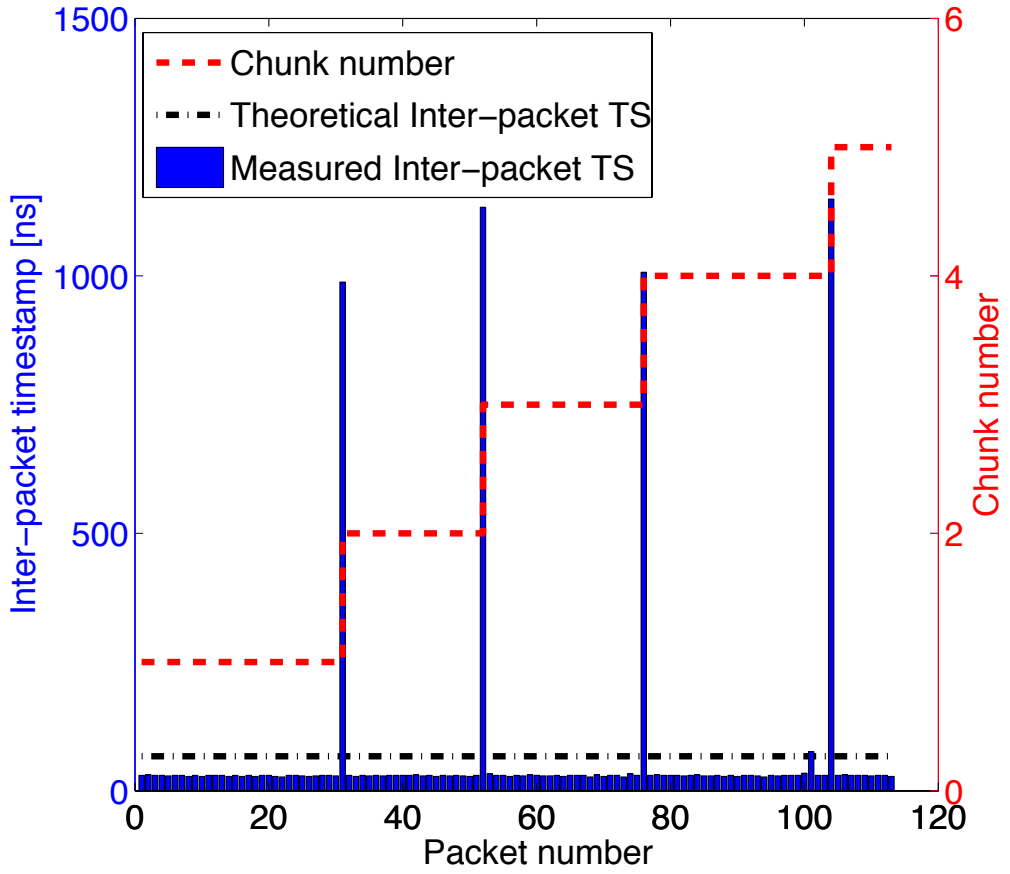


Figure 4.1: Batch timestamping

Although Linux can timestamp packets with sub-microsecond precision by means of kernel `getnstimeofday` function, drift correction mechanisms must be used in order to guarantee long-term synchronization. This is out of the scope of this chapter as it has already been solved by methods like Network Time Protocol (NTP), LinuxPPS or PTP [LCSR11].

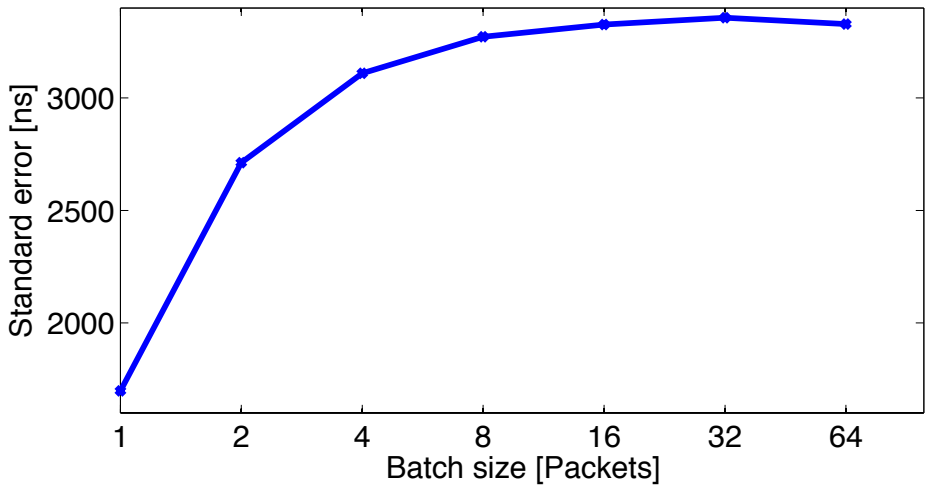


Figure 4.2: Accuracy timestamp degradation with batch size

4.3 Overcoming Batch Timestamping Issue

To overcome the problem of batch timestamping, we propose three techniques. The first two ones are based on distributing the inter-batch time among the different packets composing a batch. The third approach adopts a packet-oriented paradigm in order to remove batch processing without degrading the capture performance.

4.3.1 UDTS: Uniform Distribution of TimeStamp

The simplest technique to reduce the huge time gap between batches, called UDTS, is to uniformly distribute inter-batch time among the packets of a batch. Equation 4.1 shows the timestamp estimation of the i-th packet in the $(k+1)$-th batch, where $t_m^{(j)}$ is the timestamp of the m-th packet in the j-th batch and n_j is the number of packets in batch j.

$$\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{i}{n_{k+1}} \qquad i \in \{1, \ldots, n_{k+1}\} \tag{4.1}$$

As shown in Figure 4.3, this algorithm performs correctly when the in-

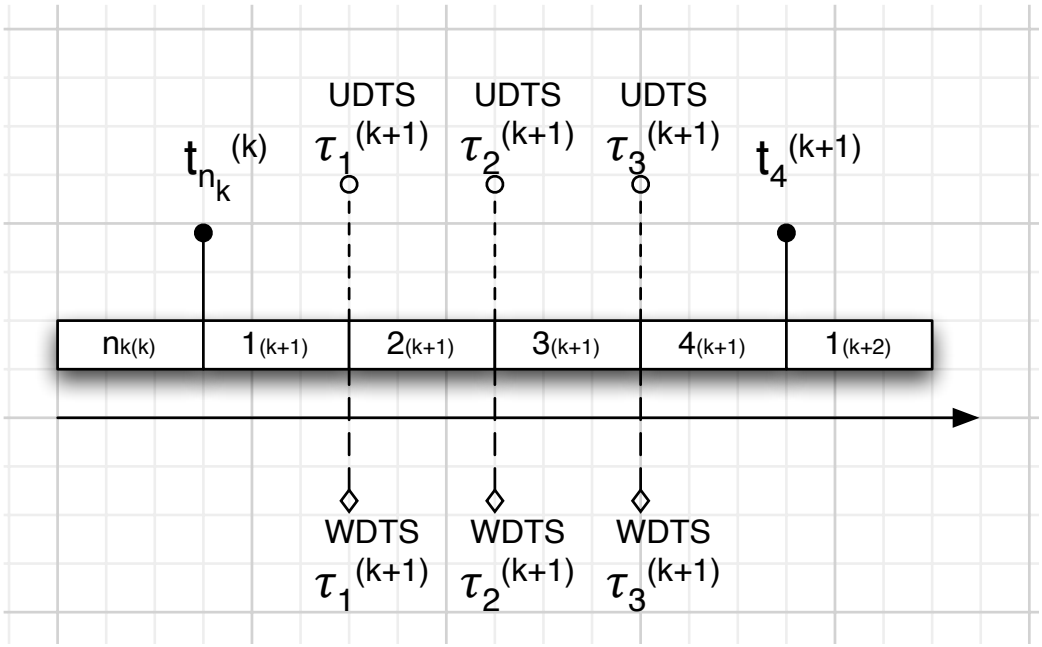


Figure 4.3: Full-saturated link with constant packet size

coming packets of a given batch have the same size. A drawback of this solution is that all packets of a given batch have the same inter-arrival time regardless of their size (see Figure 4.4). Note that the inter-packet gap is proportional to the packet size when transmitting packets at maximum rate.

4.3.2 WDTS: Weighted Distribution of TimeStamp

To overcome the disadvantage of the previous solution, we propose to distribute time among packets proportionally to the packet size. This technique is called WDTS. Equation 4.2 shows the timestamp estimation using this approach, where $s_j^{(k+1)}$ is the size of the j-th packet in the $(k+1)$-th batch.

$$\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{\sum_{j=1}^{i} s_j^{(k+1)}}{\sum_{j=1}^{n_{k+1}} s_j^{(k+1)}} \qquad (4.2)$$
$$i \in \{1, \ldots, n_{k+1}\}$$

WDTS is especially accurate when the link is completely loaded because

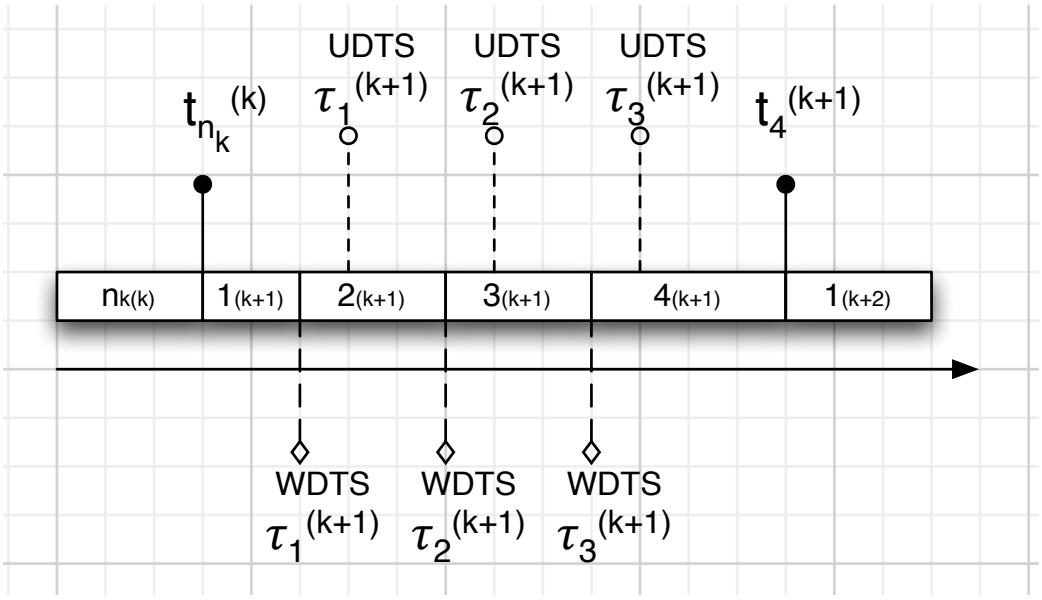


Figure 4.4: Full-saturated link with variable packet size

there are no inter-packet gaps (excluding transmission time), regardless the packet size is variable, as shown in Figure 4.4. However, when the link load is lower, both UDTS and WDTS present poorer results as they distribute real inter-packet gaps among all the packets in the batch (see Figure 4.5). That is, the lower the inter-packet gap is, the higher the accuracy is.

4.3.3 KPT: Kernel-level Polling Thread

Towards a timestamping approach that performs properly regardless the link load, we propose a redesign of the network driver architecture, called KPT. Novel packet capture engines fetch packets from the NIC rings only when a high-level layer polls for packets, then they build a new batch of packets and forward it to the requestor. This architecture does not guarantee when will the fetcher thread be scheduled and consequently, a source of uncertainty is added to the timestamping mechanism.

Our proposal is to create a kernel thread per each NIC's receive queue that will be constantly polling their corresponding receive descriptor rings,

reading the first available descriptor flags to check whether it has already been copied into host memory. If the poll thread detects that there is one or more available packets at the receive ring, they will be copied in a 1-by-1 basis to the poll thread's corresponding circular buffer. Just before each packet copy is made, the poll thread will probe for the system time by means of the Linux kernel `getnstimeofday()` function. KPT approach is implemented in HPCAP engine, described in Section 3.4.6.

A high-level application will request the packets stored in the kernel buffer by means of `read` calls over a character device file, but the timestamping process will no longer be dependent on when applications poll for new packets. This approach reduces the scheduling uncertainty as the thread will only leave execution when there are no new incoming packets or a higher priority kernel task needs to be executed. KPT causes a higher CPU burden (according to the packet rate) due to its busy waiting approach, but it does not degrade the performance to the point that packets are lost. Specifically, when receiving a real Tier-1 trace at line-rate, KPT presents a CPU load of 75% whereas the load is 40% with PacketShader [SdRRG+12]. Nevertheless, all capture engines fully occupy the CPU when receiving small sized packets—e.g., 60-byte packets.

4.4 Performance Evaluation

In this section, we assess the timestamp accuracy of our three proposals and compare them with previous approaches. Such evaluation is performed using both synthetic traffic and real traces.

4.4.1 Experimental Testbed

Our setup consists of two servers (one for traffic generation and the other for receiving traffic) directly connected through a 10 Gb/s fiber-based link. The receiver has two six-core Intel Xeon E52630 processors running at 2.30 GHz with 124 GB of DDR3 SDRAM. The server is equipped with a 10 GbE Intel NIC based on 82599 chip, which is configured with a single RSS queue to

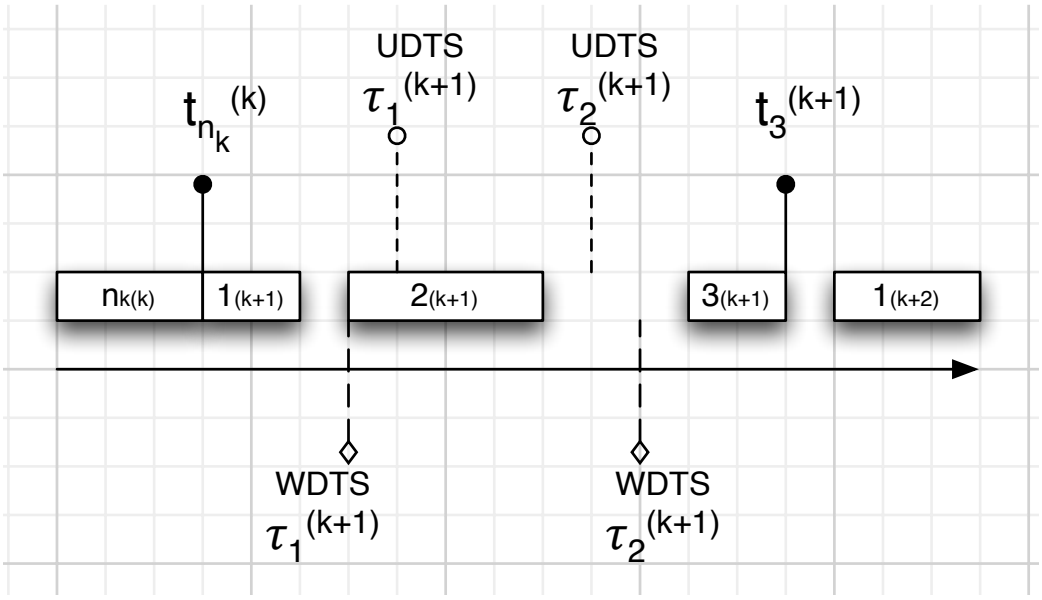


Figure 4.5: Non full-saturated link with variable packet size

avoid multi-queue side-effects, such as reordering or parallel timestamping. The sender uses a HitechGlobal HTG-V5TXT-PCIe card which contains a Xilinx Virtex-5 FPGA (XC5VTX240) and four 10 GbE SFP+ ports. Using a hardware-based sender guarantees accurate timestamping in the source. For traffic generation, two custom designs have been loaded allowing: (i) the generation of tunable-size Ethernet packets at a given rate, and, (ii) the replay of Packet Capture (PCAP) traces at variable rates.

4.4.2 Synthetic Traffic

As first experiment, we assess the timestamp accuracy sending traffic at maximum constant rate. Particularly, we send 1514-byte sized packets at 10 Gb/s, i.e., 812,744 packets per second and measure the interarrival times in the receiver side. Table 4.1 shows the error of the measured timestamp (i.e., the difference between the original and the observed interarrival times), in terms of mean and standard deviation, for a 1-second experiment for the different reviewed methods. Note that the lower the standard deviation is,

Table 4.1: Experimental timestamp error (mean and standard deviation). Synthetic traffic: 1514-bytes packets

Solution	**Batch size**	$\bar{\mu} \pm \bar{\sigma}$ **[ns]**
User-level batch TS	1	1.77 ± 1765.37
	32	1.76 ± 3719.82
Driver-level batch TS	1	1.77 ± 1742.09
	32	1.77 ± 3400.72
PFQ	-	$1.68 \pm 13,558.65$
UDTS	32	1.78 ± 167.00
WDTS	32	1.77 ± 170.95
KPT	-	1.77 ± 612.72

the more accurate the timestamping technique is. The first two rows show the results for PacketShader, chosen as a representative of batch-based capture engines. We tested with different batch sizes and different timestamping points: at user-level or at driver-level. PFQ results are shown in the following row whereas the three last ones show the results of our proposed solutions. It can be observed that timestamping error grows with batch size, as shown in Figure 4.2. However, even in the best case (using one-packet batches), the error is greater than the one observed using our proposals. UDTS and WDTS methods enhance the accuracy, decreasing the standard deviation of the timestamp error below 200 ns. Both methods present similar results because all packets have the same size in this experiment. KPT technique reduces the standard deviation of the error up to ∼600 ns. Despite timestamping packet-by-packet, PFQ shows a timestamp standard error greater than 13 μs.

4.4.3 Real Traffic

In the next experiments, we evaluate the different techniques using real traffic from a Tier-1 link (i.e., a The Cooperative Association for Internet Data Analysis (CAIDA) OC192 trace [WAcA09]). We perform two experiments:

Table 4.2: Experimental timestamp error (mean and standard deviation). Real traffic: Wire-speed and Original speed

Solution	**Wire-Speed** $\bar{\mu} \pm \bar{\sigma}$ **[ns]**	**Original Speed** $\bar{\mu} \pm \bar{\sigma}$ **[ns]**
Driver-level batch TS	13.00 ± 3171.46	$-25.95 \pm 19,399.08$
UDTS	11.88 ± 608.75	$-39.83 \pm 13,671.08$
WDTS	5.31 ± 111.22	$-41.77 \pm 14,893.97$
KPT	-1.43 ± 418.42	-43.44 ± 1093.16

in the first one, the trace is replayed at wire speed (that is, at 10 Gb/s), and then, we replay the trace at the original speed (i.e., at 564 Mb/s, respecting inter-packet gaps). Due to storage limitation in the FPGA sender, we are able to send only the first 5,500 packets of the trace. Table 4.2 shows the comparison of the results for our proposals and the driver-level batch timestamping. We have used a batch size of 32 packets because 1-packet batches do not allow achieving line-rate performance for all packet sizes. In wire-speed experiments, WDTS obtains better results than UDTS due to different sized packets in a given batch. When packets are sent at original speed, WDTS is worse than KPT because WDTS distributes inter-packet gap among all packets. This effect does not appear at wire-speed because there is no inter-packet gap (excluding transmission time). In any case, driver-level batch timestamping presents the worst results, even in one order of magnitude.

4.5 Summary and Conclusions

Batch processing enhances the capture performance of I/O engines at the expense of packet timestamping accuracy. We have proposed two approaches to mitigate timestamping degradation:

(i) UDTS/WDTS algorithms that distribute the inter-batch time gap among the different packets composing a batch

(ii) A redesign of the network driver, KPT, to implement a kernel-level

thread which constantly polls the NIC buffers for incoming packets and then timestamps and copies them into a kernel buffer one-by-one.

In fact, we propose to combine both solutions according to the link load, i.e., using WDTS when the link is near to be saturated distributing timestamp in groups of packets and, otherwise, using KPT timestamping packet-by-packet. We have stress tested the proposed techniques, using both synthetic and real traffic, and compared them with other alternatives achieving the best results (standard error of 1 μs or below). Our results, using both synthetic traffic and real traces, highlight the significant timestamping inaccuracy added by novel packet I/O engines, and show how our proposals overcome such limitation. These proposals allow us to capture correctly timestamped traffic for monitoring purposes at multi-10Gb/s rates by means of several 10 Gb/s NICs or multi-queue processing for faster interfaces such as 40 Gb/s.

To summarize, we alert research community to timestamping inaccuracy introduced by novel high-performance packet I/O engines, and proposed two techniques to overcome or mitigate such issue.

Chapter 5

Real Traffic Monitoring Systems: Statistical Classification and Anomaly Detection at Line-Rate

Previously, we showed that packet sniffing (Chapter 3) and accurate timestamping (Chapter 4) at 10 Gb/s on OTS systems is currently a fact. However, such challenges are devalued if upper applications are not able to process all packets at such high rates. In this chapter, we assess (and, actually, show) the feasibility of various monitoring tasks at high-speed rates using OTS systems. Traffic classification is our first goal. We present a statistical classification engine, able to process traffic at wire-speed. We thoroughly analyze important aspects of the flow manager (e.g., hash functions and efficient data structures), as well as the comparison of multiple state-of-the-art ML tools. Once showed the feasibility of traffic classification, we consider the problem of flexibility and scalability. That is, we wonder if it is feasible a modular monitoring system able to obtain and process network traces from different levels and granularities at line-rate. Thus, we propose an architecture as well as a monitoring application implemented over the proposed system, able to provide statistics, report alarms and afterwards perform forensic analysis.

5.1 Introduction

As shown in previous chapters, nowadays, OTS systems, based on open-source software and commodity hardware, are able to sniff and accurately timestamp packets at line-rate (10 Gb/s). The keys to achieve such high performance are efficient memory management, low-level hardware interaction and programming optimization. Unfortunately, this has transformed network monitoring into a non-trivial process composed of a set of sub-tasks, each of which presents complicated configuration details. The adequate tuning of such configuration has proven of paramount importance given its strong impact on the overall performance.

However, such capabilities (sniffing and timestamping) are not enough to perform network monitoring. That is, upper-level applications (such as a NIDS) must be able to process the captured and timestamped packets. Thus, we turn our attention to the challenges that application developers face by using this new paradigm in upper-level monitoring applications. The discovered limitation and bottlenecks in packet capturing and their respective solutions may be also useful for such upper layers.

Particularly, we focus on traffic classification. Traffic classification technology has gained importance in the recent years, as it has proven useful in many tasks such as QoS enforcement, accounting and billing, security and network management. We note that the most state of the art has focused on providing accuracy only, regardless of the processing power that is required, which may impair the practical applicability of the traffic classification algorithm in a real-world, high-speed environment. In this chapter, in order to show the feasibility of traffic classification on OTS, we present a classification engine based on open-source software and commodity hardware, able to process packets at line-rate.

Likewise, the heterogeneity and high complexity of current networks (large number of different applications and protocols, aggregates of multi-Gb/s, millions of packets per second and millions of concurrent flows per link) call for more flexible and scalable monitoring designs. Such designs must be capable of processing network data at different granularities (packet-level, flow-level,

and aggregated statistics) with different purposes (e.g., anomaly detection and traffic classification) at line-rate. Consequently, in this chapter, we propose a modular architecture to build high-performance OTS monitoring systems fulfilling flexibility and scalability requirements.

The rest of the chapter is organized as follows: on the one hand, Section 5.2 is devoted to present and analyze our proposed classification engine. Particularly, Sections 5.2.2 and 5.2.3 describe the system architecture and the proposed configurations, respectively. In Section we stress test the proposal in a worst-case scenario whereas in Section 5.2.6 evaluate it in a real scenario. Finally, Sections 5.2.7 and 5.2.8 thoroughly analyze flow handling and classification modules, respectively. On the other hand, Section 5.3 presents a system architecture to overcome high-speed scalable network monitoring and further provides a real sample, DetectPro, a line-rate flexible passive probe. In particular, Section 5.3.2 describes the proposed architecture and Section 5.3.3 shows the applicability of such proposal with a sample. In Section 5.3.4 we detail the experimental procedure to assess our proposal whereas the performance evaluation results are shown in Section 5.3.5.

5.2 *HPTRAC*: Wire-Speed Early Traffic Classification Based on Statistical Fingerprints

5.2.1 Introduction

In this Section, we present an open-software-based traffic classification engine, called High-Performance TRAffic Classifier (HPTRAC), running on commodity multi-core hardware, able to process in real-time aggregates of up to 14.2 Mpps over a single 10 Gb/s interface – i.e., the maximum possible packet rate over a 10 Gb/s Ethernet link given the frame size of 64 Bytes. We perform a thorough sensitivity analysis involving important aspects of the system, such as the use of specific hash functions and efficient data structures for flow management, as well as the use of multiple state-of-the-art ML tools.

Hence, we argue that commodity multi-core hardware offers intrinsic scal-

ability at low cost, while providing the unbeatable flexibility of software-only solutions. Hence, we take a different twist with respect to works employing specialized hardware based on Network Processor or FPGAs: furthermore, our software based solution is the first to achieve two important milestones. First, using both real Tier-1 traces and synthetic traffic, we demonstrate that multi-Gb/s statistical traffic classification is feasible with OTS systems. Second, our solution is able to sustain higher classification rates than previous work [SGV+10, LXSL08, VPI11, LKJ+10] with a sizeable gain in terms of the maximum amount of classification actions per second and manageable packet rates.

In more detail, our software can easily handle a real Tier-1 traffic aggregate (i.e., a CAIDA OC192 trace [WAcA09]) replayed at 10 Gb/s, corresponding to 1.6 Mpps and 58 thousand flow classifications per second (Kfps). Using two interfaces, our system sustains classification rates of 20 Gb/s, 3.2 Mpps, 116 Kfps. Yet, the upper bound of the system performance is much higher, as we manage to handle classification rates up to 14.2 Mpps and 2.8 Mfps without any losses (benchmark with synthetic worst-case traffic scenario with trains of 64B packets, 5 packets per flow, over a 10 Gb/s link).

Such astonishing performance follows the use of a lightweight classification algorithm (which base their decisions upon the size of the first 4 packets of every unidirectional flow [BTS06, CDGS07]), as well as of the most recent advances in terms of packet processing techniques [HJPM10]. Yet, as we will see in the following, engineering the system so that it sustains wire-speed classification in the worst-case traffic scenario required investigation of several delicate architectural trade-offs of the software, as well as the careful tuning of hardware parameters. We believe the removal of all software bottlenecks in the workflow to be another major contribution of this section.

This section brings many additional results, notably a thorough sensitivity analysis involving both important flow management aspects (such as the use of specific hash functions and efficient data-structures for flow management) in 5.2.7, as well as the use of multiple machine learning tools (e.g., Naïve-Bayes, C4.5, Support Vector Machine (SVM)) in 5.2.8. In addition, we carried out an extensive and intensive stress test of the capture module,

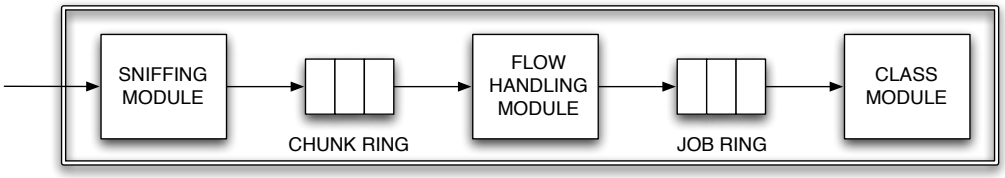


Figure 5.1: HPTRAC modules

i.e., receiving traffic at 14.2 Mpps for a 24-hour span, and a detailed profiling of the whole system.

Finally, we note that to stress-test our system, we also had to develop highly efficient traffic injection engines able to either generate infinite synthetic traffic or replay very long real traces thus saturating 10 Gb/s links for extended time window. As an additional contribution of this thesis, we make the developed tools, both the classification engine and the traffic generator [Hrg12], available as open-source software as a research byproduct.

5.2.2 System Architecture

We report in Figure 5.1 the three main blocks that compose our classification system: the two data ring structures for packet/flow queuing and the logical connections that push information from left to right.

Sniffing module

We capture incoming packets with PacketShader [HJPM10], a customized version of the Intel ixgbe driver that can fetch chunks of multiple frames from the NIC to user-space, greatly reducing the I/O overhead and the per-packet buffer allocation cost. Thanks to a native feature of the Intel 82599 10 GbE controller, incoming frames are partitioned in RSS queues according to a hash function: one sniffing module (a thread running in user-space but accessing with zero-copies to packets in kernel-space) can then be set up to fetch frames only from a given RSS queue. Following such paradigm, we carefully tune the system affinity. That is, interruptions of a given RSS queue are handled by a specific core (interrupt affinity) and every capture thread is

tied to the same CPU core (thread affinity) so as to keep the data locally in that CPU cache (hence limiting cache thrashing between processor sockets) and to avoid context changes. This same feature extends parallelism from the NIC to the user layer, as RSS queues feed different cores with multiple chunks at the same time, pushing them through multiple lanes of the PCIe bus and hence increasing the overall throughput with respect to a single core solution. See Chapter 3 for more details.

Packets are then organized in a circular ring and made available to user-space with zero-copy technology. Here a thread running on the same CPU copies the chunks from the kernel ring and enqueues resulting data to a *Chunk Ring* of Figure 5.1: if the ring is full, a chunk might be lost. We set the chunk size to 128 packets.

Flow Handling module

A thread then dequeues packets from the Chunk Ring and performs lookup into a *Flow Table*. A hash over the packet 5-tuple is used as a primary key to access a hash table, while collisions are handled by chaining (a data structure based on linked list of flow buckets). Once the bucket is found (or a new one is appended if the flow was not already known), a new feature is added to the flow structure, namely the length of the corresponding packet (read from the IP header). Each flow is considered active within a timeout (default 15 seconds) after the reception of the last packet. When the timeout expires, its position in the linked list can be reused by a new flow (no deallocation overhead). Once a configurable number of packets for a given flow has been seen (4 in this work), a new classification job is fired to the *Job Ring* of Figure 5.1 if a position is available (otherwise, the job will likely be inserted the next time a packet from that flow will be analyzed). We set the flow hash table size to 54 millions.

We report an exhaustive sensitivity analysis of the flow manager (e.g., hash table size and hash functions) in Section 5.2.7.

Classification Module

Classification threads are able to run custom implementations of several statistical classifiers, such as Naïve-Bayes, C4.5 trees and SVM. Given the first four packets of a flow have been received, the algorithm associates the flow to the protocol whose model scores the maximum likelihood for the generation of the flow. For each protocol model the algorithm uses the size of the packets as indexes into the four lookup tables that have been associated to that protocol during the training phase: the values extracted are then summed together, we optimize, in fact, the algorithm by storing the logarithms of the table values to avoid products as reported in [RVV+08]. By comparing the values obtained for each of the protocols for which a model is available, the algorithm chooses the application and the classification of the flow terminates: for more details please refer to [CDGS07]. Although classification accuracy, in terms of packets *correctly* classified, is a key issue, it is not the goal of this study because it has been already analyzed [NA08]. Once chosen the classification technique, we evaluated its performance in terms of computational cost and the feasibility of its implementation on a real system (based on commodity hardware and open software). Note that the computational complexity, given a model, is not a function of accuracy.

To increase the speed at which Jobs are extracted from the Job Ring, multiple threads can be spawned according also to the complexity of the algorithm. Each thread extracts one job from the ring, processes the data, and writes the classification verdict in the flow bucket inside the Flow Table. New packet for that flow will be marked with this verdict in their ToS field of the IPv4 header. By default, we set only one thread to classify: as we will see in the experimental section, this is sufficient to classify all jobs generated by the flow manager (for all machine learning algorithms, except SVM). Hence, unless otherwise stated, we use a Naïve-Bayes classifier, and we defer a detailed comparison of several state-of-the-art machine learning approaches to Section 5.2.8.

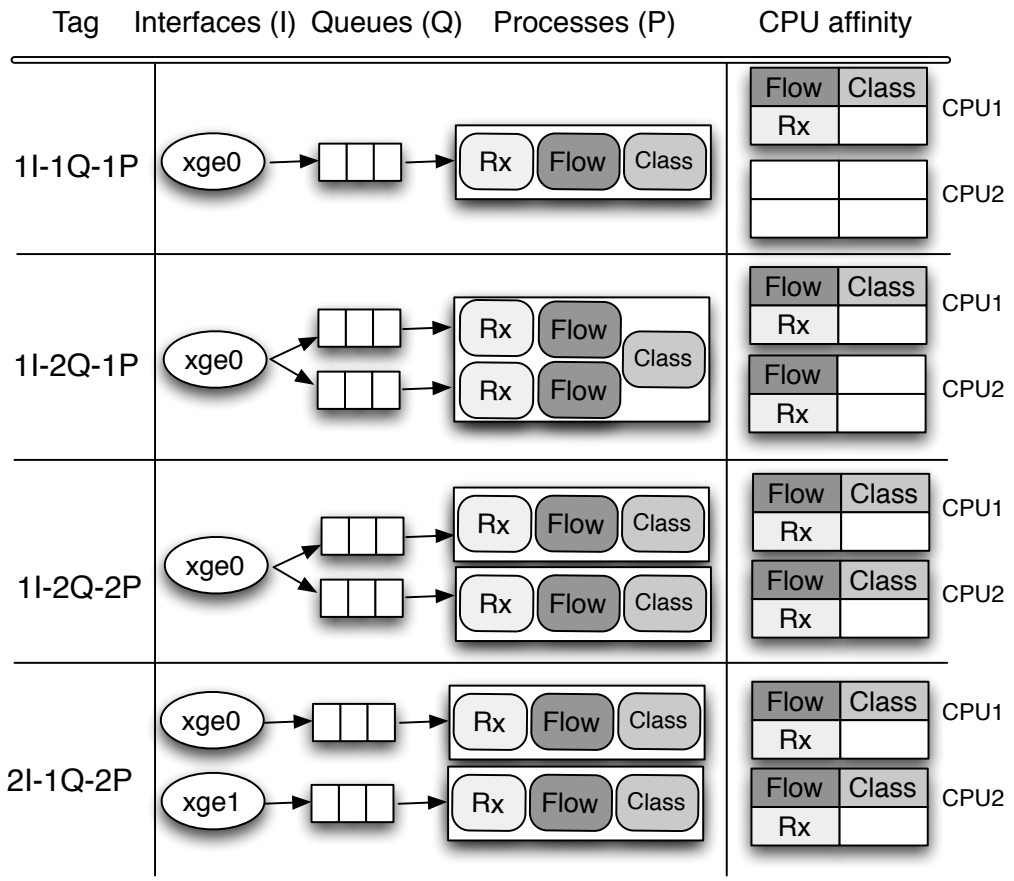


Figure 5.2: Different architectural configurations of HPTRAC system

5.2.3 System Configuration

While even the most simple module configuration is able to sustain the real OC192 traffic replayed at 10 Gb/s, careful engineering is needed to achieve the highest classification rates under worst-case traffic (i.e., smallest packets and short flows). Configurations explored in this analysis are sketched in Figure 5.2.

1I-1Q-1P

This is the simplest configuration: traffic received at the same RSS queue of a single interface is captured by a thread, which in turn pushes packets

to one chunk ring in user-space. All packets are matched in the flow table sequentially: one classification thread works on the single job ring. *In this case, though, some CPU cores are not utilized.* Note that using a single RSS queue may be preferable to multi-queue in some cases. Namely, whenever a single CPU is enough to cope with line-rate processing, the lower CPU usage translate into a lower power consumption and thus carbon footprint, not to re-implement monitoring tools which have not been designed for parallel processing, or to avoid packet reordering issues due to multi-queue [WDC11].

1I-2Q-1P

This configuration holds the single process model of the previous one but two RSS queues are used: this means two threads for fetching packets chunks and two separate flow matching modules. Each capturing flow is executed on a different CPU, but only one hash table is used. Since a single job ring is used, data coming from two different CPUs is merged again in a single data flow. *In this case, locking issues may arise on the flow table data structure.*

1I-2Q-2P

Though similar to the previous configuration (two RSS queues, two threads for capturing, bound to different CPUs), the two threads for flow-handling *residing in different processes*, each on the same CPU of the corresponding sniffer. This means that two separated flow tables are maintained and no locking is required thanks to the hash function used at the NIC for dividing packets into the two queues: each single flow lives on a single queue only (no mixing). Moreover, the classification code (threads) and data structures (job rings) are duplicated and fairly spread between the two CPUs with no data flow merge. *In this case, locking is solved at the price of doubling the amount of memory.*

2I-1Q-2P

The last configuration is exactly as the first one, but two NICs are used: for each NIC a complete capture and classification chain is instantiated, each

complete chain lives on a separated CPU. *Given the number of cores in our system, we cannot explore other configurations when 2 interfaces are in use.*

5.2.4 Hardware and Software Setup

First, let us describe our experimental testbed, covering hardware and software details.

Hardware setup

Our setup consists of two general-purpose servers: one acts as traffic generator, the other receives and classifies the traffic. Both are equipped with 2x12 GB DDR3 SDRAM memory boards and features two Intel Xeon E5620 processors, counting four cores each (with hyper-threading capabilities disabled to obtain actual parallelism among cores) working at 2.40 GHz. Concerning connectivity, each server is equipped with one dual port 10 GbE Intel X520-SR2 NIC, and servers are directly connected with a fiber link. This NIC model is based on 82599 chipset, which allows multi-queue techniques, up to 16 RSS queues per interface and direction.

Software

Both servers have installed Ubuntu 10.04 server 64-bit version, using 2.6.35 Linux kernel. In order to send traffic we cannot utilize a generic tool such as tcpreplay [Tcp12] (which allows to replay a packet-level trace) since it cannot saturate 10 Gb/s links. For this reason, we have developed a tool that uses PS API to send traffic at maximum rate. Particularly, our tool [Hrg12] is able either to inject infinite synthetic traffic or to replay very long packet-level traces.

5.2.5 Stress Testing

In this Section, we benchmark system performance in a worst-case scenario (64B packets at maximum rate) in order to locate and solve system bottlenecks using synthetic traffic. Synthetic traffic consists of TCP segments

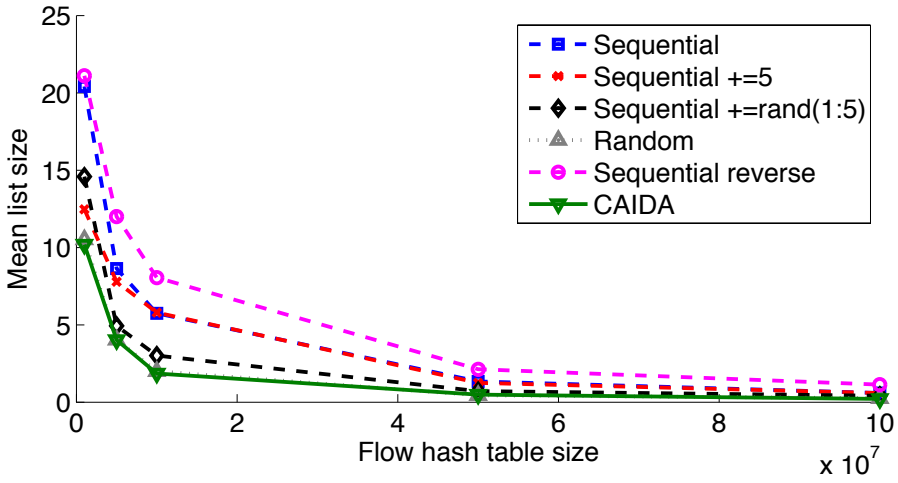


Figure 5.3: Flow hash table list occupancy for different traffic pattern

encapsulated into 64B-size Ethernet frames, forged with incremental IP addresses and TCP ports. Incremental TCP and IP pattern may seem a best-case pattern in terms of hash collision. However, we did several tests with different patterns (sequential in various order, random, 5-tuples of real traces) and the results were similar—Figure 5.3 shows the mean list size of flow hash table entries after processing 840 M packets with a 30-second expiration timeout. Note that 14.2 Mpps lling a 50 M sized hash table produces a sizeable amount of collisions.

We stress once more that, though all packets have 64B-size, the packet length features are extracted from the IPv4 header: hence, our benchmark methodology do not affect the relevance of the classification results. For each flow (5-tuple combination), we send 5 packets, since the 5-th packet will be the first to have the chance to be classified (on the basis of the packet-length features of the previous 4 packets) At a maximum rate of 14.2 Mpps for 64B frames, this translates into 2.8 Mfps.

We first stress test the system in a worst-case scenario, i.e., using synthetically generated 5-packets long flows, sending 64B frames at maximum rate, i.e., 14.2 Mpps or 2.8 Mfps per interface. We measure the amount of packets processed by each module during 60-second experiments. Note that using two interfaces we only were able to send at ≈25 Mpps (instead of the

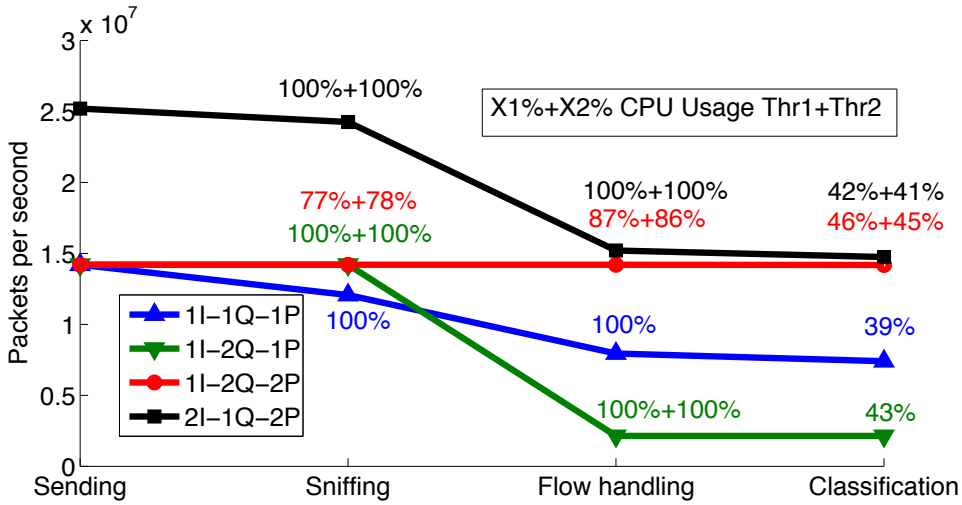


Figure 5.4: HPTRAC performance. Worst-case scenario: synthetic traffic 64B packets, 5 packet/flow

theoretic maximum 28.4 Mpps) due to limitations in the sender.

Figure 5.4 shows the performance obtained for each module (namely, sniffing, flow handling and classification) for the different configurations. In the parallel coordinates plot, a negative slope in the curves reflects a bottleneck: i.e., a module that is not able to process all packets generated by the previous one. Curves in Figure 5.4 are annotated with the CPU usage of each module; when the configuration has two threads in one module, CPU usage is expressed like an addition of two terms. CPU usage is computed using `sysstat` utilities [God12]. We obtained the CPU load per thread every 5 seconds and then we averaged. Background CPU usage (when there is no classication systems running) and experimental variance are negligible.

The simplest configuration.

The simplest configuration 1I-1Q-1P sniffs traffic from one interface, uses one RSS queue and one process. In this case, it can be observed that not all packets sent can be sniffed using a single RSS queue, and therefore a single core. Particularly, only 12.1 Mpps are sniffed out of the 14.2 Mpps sent: note that sniffing module CPU usage is 100%, which pinpoints a processing power

bottleneck. Similarly, flow module is only able to process 7.9 Mpps out of 12.1 Mpps sniffed packets. As CPU utilization of flow handling module is also 100%, we have strong indication of a second processing bottleneck. Finally, a slight negative slope can be observed in the classification module. This is not due to a CPU bottleneck (39%), but rather to the fact that only flows with 5 packets can be classified and there have been packet losses in previous modules. The classification rates sustained by 1I-1Q-1P configuration are thus 7.9 Mpps and 1.5 Mfps.

Using 2 RSS queues

In order to remove bottlenecks observed in the first configuration, we increment the number of RSS queues and the number of cores dedicated to packet sniffing and flow management. Thus, we test two configurations, namely 1I-2Q-1P and 1I-2Q-2P. As shown in Figure 5.2 and explained in Section 5.2.3, the former configuration uses one process with two threads for sniffing, one per queue, and two threads for flow handling, having a unique hash table and a unique classification thread. Conversely, 1I-2Q-2P configuration uses two processes, one per queue, which do not share neither data structures nor processing cores. We can observe that the bottleneck in the sniffing module is removed in both cases: i.e., two RSS queues are enough to receive and copy to the chunk ring at a 14.2 Mpps rate. Besides, the 1I-2Q-2P configuration with two processes consumes less CPU power.

Locking issue

The behavior of the flow handling module is different between 1I-2Q-1P and 1I-2Q-2P. Indeed, 1I-2Q-1P is not able to process all packets received by the sniffing module, and performance even worse than in the case with only one RSS queue. The bottleneck in this configuration is tied to the contention in the access to shared data structures, such as the hash table and the job ring. To arbitrate concurrent access to shared memory is necessary to perform synchronization and locking operations, which cause a huge performance loss. As in the first configuration, all flows generated by the flow handling module

can be classified. With 1I-2Q-1P configuration, the performance of the whole system falls to 2.1 Mpps and 0.4 Mfps.

Wire-speed classification

Using two RSS queues and two independent processes, we remove both sniffing bottleneck and flow management locking issues. With 1I-2Q-2P, the system sniffs, process and classifies all packets sent at wire-speed without packet losses. Notice further that not even a single CPU core is saturated (sniffing 77%+78%, flow management 87%+86%, classification 45%+46%), so that the remaining processing power could be useful to perform other tasks (such as packet forwarding or statistics collection). This also means that the processing capabilities of 1I-2Q-2P configuration exceed the maximum data rate at 10 Gb/s., i.e., 14.2 Mpps and 2.8 Mfps.

Using 2 interfaces

The latest configuration, 2I-1Q-2P uses two interfaces to receive traffic, but a single RSS queue per interface due to the limit in the number of cores. As expected, behavior is similar to 1I-1Q-1P, with bottlenecks in both sniffing (24.2 Mpps received out of 25.2 Mpps sent) and flow handling (15.2 Mpps processed out of 24.2 Mpps received).

More extensive tests

As we observed in Figure 5.4 configuration 1I-2Q-2P is able to process all packets sent at 14.2 Mpps without packet losses. However, the experiment span is 60 second. To assure that our system is packet-loss-immune in the long-term, we assess the sniffing module sending traffic for 24 hours. The results show that it only dropped 60 thousand packets out of 1.2 trillion packets sent during the whole experiment, corresponding to an average packet loss rate of $4.6 \cdot 10^{-8}$. To remove this almost negligible packet loss rate, we set a higher priority for our system processes in the operating system scheduler. Particularly, we set a -20 nice level, the highest priority level in a Linux system. With such priority level, we obtained zero packet loss.

Table 5.1: HPTRAC profiling: Top-5 most consuming-time functions

Function Name	Module	Time (%)
`processPkt`	Flow-handling	73.91
`capture`	Sniffing	18.78
`readPktFromChunkRing`	Flow-handling	3.65
`flowHash`	Flow-handling	1.34
`class`	Classification	1.28
Subtotal	-	98.96

Profiling

In order to locate more precisely which are the most time consuming functions and where we should tune our system to improve performance, we profiled our system using `gprof` [FS93], for a 5-minute experiment. Table 5.1 shows the percentage of time spent for each function. We do not consider initialization functions, e.g. hash table allocation. Only top-5 most time-consuming functions of our system are shown, which correspond to 98.96% of the total time. It can be observed that packet processing function in the flow manager, which is mainly dedicated to hash lookups, spends almost 75% of the time, whereas flow hash computation spends less than 1.5%. Previous results suggest that flow manager is the potential bottleneck in our system. Thus, we exhaustively assess data structures and hash functions in order to better understand the behavior of the flow-handling module, in Section 5.2.7.

5.2.6 Performance Evaluation in a Real Scenario

In this Section, we assess our system in a real scenario, replaying traces from a Tier-1 link over one or two 10 Gb/s interfaces. Real traffic consists of a packet-level trace sniffed in 2009 at an OC192 (9953 Mb/s) backbone link of a Tier-1 ISP located between San Jose and Los Angeles, available from CAIDA [WAcA09]. All the packets in the trace are anonymized and captured without payload, the average of the original packet size in the trace is 744 bytes and the average number of packets per flow is 49.

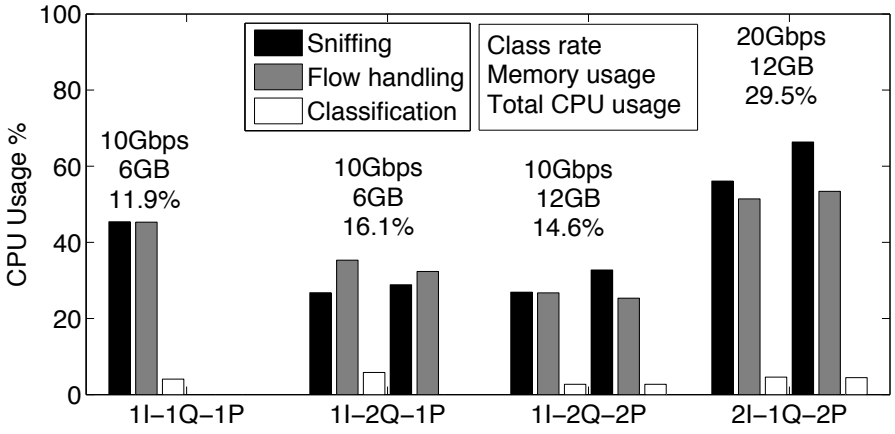


Figure 5.5: HPTRAC performance. Real scenario: CAIDA trace with original packet length

Original packet size

We now test performance on real Tier-1 traffic. We replay traces by sending packets back-to-back at 10 Gb/s by filling payload with zeros. Using full packet size in Fig5.5, all system configurations sustain maximum rate: i.e., 1.6 Mpps and 58 Kfps on a single 10 GbE interface, or 3.2 Mpps and 116 Kfps on two interfaces. CPU usage and memory occupancy report that cores are far from being saturated and, thus, our system boundaries are even beyond 20 Gb/s in a realistic scenario. Note that the simplest configuration (only three threads) is enough to classify all traffic with the smallest CPU load (hence the lowest carbon footprint).

Capped packet size

Finally, in Figure 5.6 for each frame of size S_i we control the maximum amount of bytes sent on the wire as $\max(S_i, L)$ with L the maximum frame size, that we vary $L \in [64, 1500]$B to tune the flow and packets arrival rates for a fixed datarate of 10 Gb/s – hence finding the packet and flow processing rate bottleneck of each configuration. From the figure, we gather that the simplest configuration 1I-1Q-1P, can sustain up to 3.8 Mpps and 103 Kfps using only 3 cores. When maximum packet size is 750B (or above), the

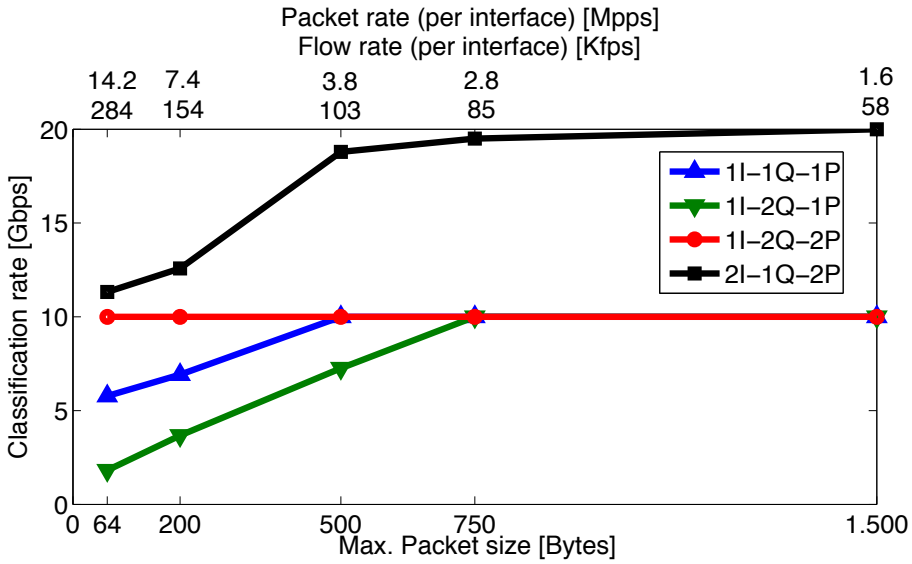


Figure 5.6: HPTRAC performance dependency on packet size. CAIDA trace with capped packet length

system is able to sustain line-rate.

5.2.7 Flow Manager Analysis

The main result from 5.2.5 was to determine the best configuration in terms of (i) interfaces, (ii) queues per interface, (iii) number of processes and (iv) threads for avoiding losses even in the worst case scenario (synthetic traffic at maximum packet rate). It turned out that two "isolated" processes are able to pick-up packets from two separated driver queues at 7.1 Mpps each. Given that traffic classification does not impact on losses because of the Job Ring filter mechanism, the bottom line is that two "isolated" processes are required to match all the received packets in the flow table. We use "isolated" to stress that the sets of flows seen by the two processes are disjoint thanks to the hardware RSS mechanism.

In this section, we focus on the flow matching subsystem, aiming at improving as much as possible the performance of each single process. To this end, we run some off-line experiments to test different flow management architectures, i.e., we change the *hash function* and the technique for *handling*

collisions in the hash table. For each experiment we pre-load IP tuples in memory from the real traces and we measure the time needed to process the packets they represent, i.e., to search the flow table, store the new tuples and remove the old ones. To run very long tests and avoid to incur in memory being swapped to disk, we break each experiment into batches, so that tuples can be stored/analyzed in/from live memory: we then sum up the overall time needed to process all the batches to draw the overall performance figures.

Hash function

We report in Figure 5.7 results for the CAIDA trace when the flow table is addressed by the hash function used in theCoral Reef (CR) software suite [MKK+01, CAI11] (continuous lines) and by the Bob-Jenkins hash that was included in the Linux Netfilter code in 2006 (dashed lines). In both cases collisions are handled by linked lists. Though the main purpose of the hash is to distribute the observed tuples uniformly across the flow table, every hash function tends to collide some tuples into the same bucket and this phenomenon can be really manifest when the number of rows in the table is close to the number of elements to be managed.

To force this situation we start by considering a CR hash table with 4,999,999 rows [1] : the CAIDA trace we use for this test is, in fact, composed of 150 Millions of packets belonging to approximately 4.3 M of flows. Despite the small table size, the corresponding line with upward-pointing triangle shaped markers displays better performance (8.5 Mpps on average) than the configuration we previously used in Section 5.2.5, where we considered a table with 54 M of rows—line with the square shaped markers, 7.1 Mpps on average.

Notice further that length of the longest chain in the small table ($L = 35$) exceeds that of the large table ($L = 33$), which follows from the fact that in the small table 2.6 M out of 4.9 M rows are used (high collision rate) while 3.4 M out of 54 M rows are used with the second one (lower collision rate).

[1] We choose this number because it is prime

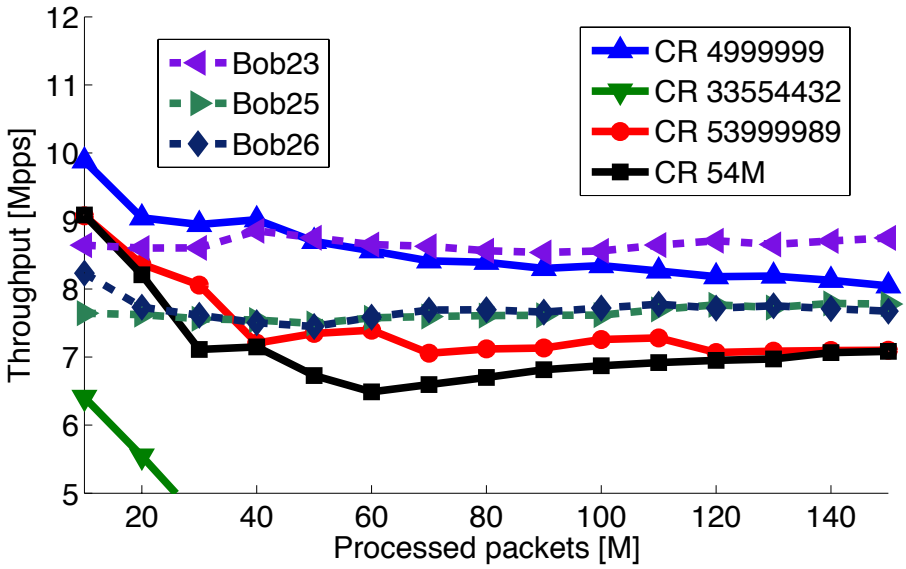


Figure 5.7: HPTRAC flow manager sensitivity to hash function

Since no swap is taking place during experiments, the only difference between the two configurations is the table memory footprint. Hence, reasons of this counter-intuitive result are likely tied to memory access issues (e.g., page faults and complex memory cache mechanism of the Xeon CPU).

Performance does not improve even if we consider a huge table with a prime number of rows (line with circle shaped markers, 7.4 Mpps on average, longest chain is still 33 elements) while collapses if the number of rows of the table is a power of two (line with downward-pointing triangles partially displayed on the bottom left of the figure, 3.5 Mpps on average, longest chain has thousands of elements). Overall, despite its low computational complexity (few simple operations including XORs and only one integer product) the CR hash leads to very frequent collisions.

The "Bob" Jenkins hash, instead, is known to never generate more collisions than predicted by the analytical bound [HSZ09]. We considered three Bob configurations, with increasing number of rows that for this hash function must be a power of two. More precisely, we consider Bob23 ($2^{23} = 8.4$ M rows), Bob25 (33.5 M) and Bob26 (67.1 M). Bob23 row exhibits the best performance (8.7 Mpps on average, line with left-pointing triangles) even though

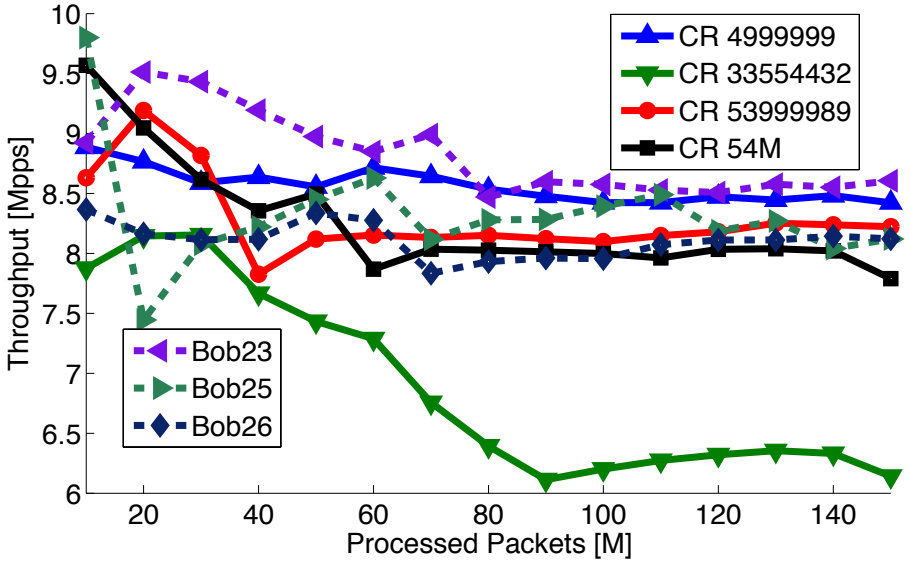


Figure 5.8: HPTRAC flow manager performance with RedBlack trees

the improvement is modest with respect to best CR: for comparison, the longest chain is 9 elements instead of 35. Again, while increasing the number of rows decreases the chain length (maximum chain is 5 for to Bob25 and 4 for Bob26), the huge memory fingerprint of the corresponding tables seems to play against performance (partially overlapping lines with right-pointing triangle and diamond shaped markers, approximately 7.7 Mpps for both).

Collision management

To relief the flow management load due to chaining, we explore Red Black (RB) trees for handling collisions in the hash table [GS78]. This data structure is known to be automatically balanced, thus guaranteeing all operations to be O($\log n$). In particular searching is much more efficient than in the linked list implementation, where this operation is O(n). To understand the appeal of RB trees it is worth to say that in our experiments, at least a search is performed for every observed packet.

We report in Figure 5.8 the performance of RB Tree alone for different hash functions, hence repeating the same analysis of the previous case. De-

Table 5.2: Performance comparison on CAIDA trace for different hash functions and data structures

Hash type	*List*		*Tree*	
	Mpps	Chain	Mpps	Chain
CR4999999	8.53	35	8.56	5.13
CR33554432	3.47	3000	6.85	11.55
CR53999989	7.42	33	8.29	5.04
CR54M	7.12	33	8.24	5.04
Bob23	8.67	9	8.82	3.17
Bob25	7.66	5	8.32	2.32
Bob26	7.71	4	8.12	2.00

tailed comparison with chaining is reported in Table 5.2. Some remarks are in order. First, Figure 5.8 shows that after a very long transient (80 M packets) switching to RB Trees compresses performance related to every hash function into a narrow region (between 8 Mpps and 8.5 Mpps). Chaining in fact, is really similar in all cases as reported in the last column of Table 5.2: with respect to the list case with the same hash function, a RB Tree structure will face the same collision rate hence switching chaining from n to $\log n$. This is really evident in the CR33554432 case where the longest list with thousands of elements is replaced with a tree of depth 12 (line with downward-pointing triangles this time fit into the figure).

Second, Bob23 and CR4999999 are still top scorer. It is worth noting however that Bob23 (after a promising start) behaves slightly worse than the analogous list implementation: if on one hand the depth of the taller tree is approximately 3 with respect to the longest list with 9 element, on the other hand the complexity in handling the tree, especially the operations for recycling old buckets, makes the list faster. With CR4999999 we observe the opposite: in this case, in fact, we reduce chaining from longest list with 35 elements to taller tree with depth 5 which is enough to balance more complex tree operations. Similar gains can be observed for the remaining CR hash functions (and, in fact, chaining reduction is similar too), while change of performance for Bob25 and Bob26 is not noticeable: very short list are, in

fact, replaced with very small tree and they are all so compact structures that they do not impact on flow matching.

Finally, we confirm the same effect previously observed: that is, structures with less rows and a shorter memory footprint perform better than larger structures. Overall, we conclude that simpler structures (i.e., list w.r.t RB trees) and smaller tables (e.g., Bob23, CR4.9M) achieve higher flow management rates.

Router position

We finally explore how performance changes if we run the same test on a complete different data set. We therefore compare the performance of core vs edge traffic classification. To do so, we consider a daylong trace captured at the Digital Subscriber Line Access Multiplexer (DSLAM) of an European customer ISP, that we cannot disclose due to Non-Disclosure Agreement (NDA). In reason of the previous section, we report in Figure 5.9 results for Bob23, separating performance of incoming and outgoing. At a glance, it can be seen that performance only slightly improve with respect to CAIDA core traces: this means that the hash functions we use correctly handle both traffic types. As for core traffic, Bob23 spreads heterogeneous tuples almost uniformly over the table. The same goes for edge traffic, characterized by a smaller tuple variability.

Conclusion

To summarize, state-of-the-art data structure (balanced RB tree) and hash functions (Bob-Jenkins hash), are not enough to let a single process manage traffic at 10 Gb/s in the worst case scenario of more than 14 Mpps on current commodity hardware. Additionally, we see that complex structures for collision management, such as RB trees, do not payoff, and that tables with a smaller memory footprint and carefully chosen hash functions should be preferred. If the trend in line rate vs. CPU speed will remain the same, this result either:

(i) Calls for completely rethinking the data structures and the hashes or

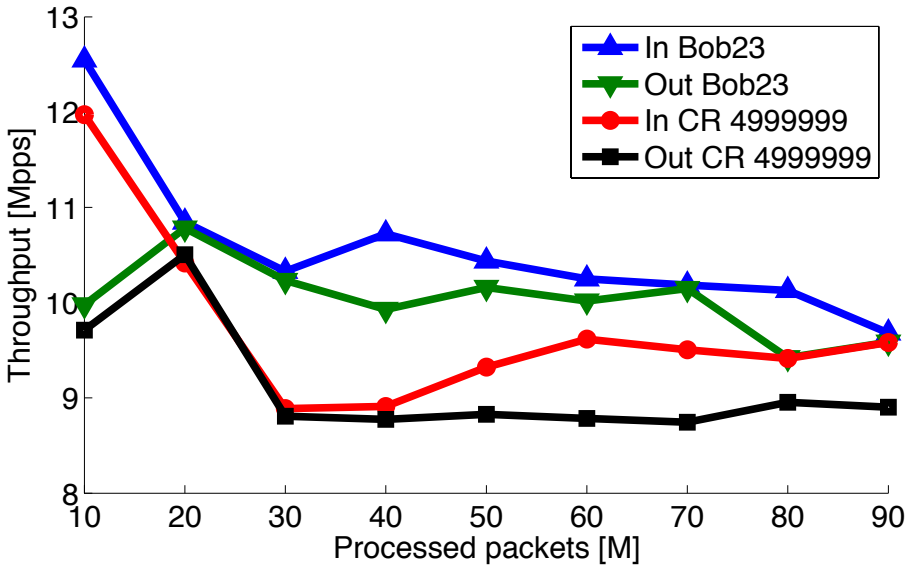


Figure 5.9: HPTRAC flow manager. Performance comparison at the edge

more likely.

(ii) Confirms that the multi-core multi-process approach outlined in this study shall be generalized to other high-performance traffic analysis applications.

5.2.8 Classification Analysis

We have implemented, evaluated and compared three representative state-of-the-art statistical classification methods, such as Naïve-Bayes, C4.5 Trees and SVM whose accuracy has been previously showed as enough for traffic classification purposes [NA08]. Once chosen the classification technique, we evaluated its performance in terms of computational cost and the feasibility of its implementation on a real OTS system (based on commodity hardware and open software).

As of course the complexity of a classifier may strongly depend on its accuracy, one might argue that we should report both to be fair. We want to underline, however, that thanks to previous works we can completely skip the accuracy evaluation: once, in fact, the validity of an algorithm is accepted,

one can also test its computational performance using fake feature vectors or models, given that the *number of features* in each vector under testing and the *number of models* remain the same. To better clarify, the computational complexity will not change if one takes an award winning algorithm and chooses different features (leading to different models) than those which improve the accuracy: for instance, if an SVM–based classifier extracts for each flow a feature vector made of the average length of all packets in the flow, the destination port number, the inter-arrival time of the first two packets and the third byte of the second packet payload (four features per vector), its complexity in finding which one out of eight models better describes the vector is exactly the same as in our system (which matches the first four packet lengths against eight models).

This not only allowed us to avoid assessing the accuracy, it also helped us in finding the system limits: we can, in fact, test worst case scenarios by injecting worst case data (very short packets) to determine the maximum throughput that our system can sustain. Though in this case accuracy would be meaningless, nevertheless our system will cope with "slower" (less packets per second) genuine traffic, for which instead accuracy will be as high as found by previous works that tested the same algorithms.

By the way, testing both computational and accuracy performance could be also tough: e.g., traces gathered at 10 Gb/s speed, like the one we use to stress test our system, do not provide a labeled ground truth as they are anonymized and have the payload stripped out. Conversely dataset [oB09] for which we have ground truth was collected on a 100 Mb/s links, and could not be representative of the traffic aggregate that shows up on a 10 Gb/s router. Luckily, thanks to the aforementioned reason we can focus on computational performance given that *the accuracy of the algorithms that we test here has already been proved*: to this end we report such results in Table 5.3. In particular, previous work [EGS09] reported in the fourth row of Table 5.3, validates the accuracy of the classification technique we use in this work (i.e., four packet lengths features in each flow-vector, with SVM classification), reporting a 93% of correct classification on average. As a general tendency, moreover, we note that accuracy constantly improved over

Table 5.3: Flow and Byte Accuracy for Early Classification Techniques

Flow	Byte	Datasets	Year	Ref
0.94	-	Campus	2006	[BTS06]
0.94	-	Unibs[oB09]	2007	[CDGS07]
0.93	-	several	2008	[KCF+08]
0.97	-	several	2010	[LKJ+10]
0.99	0.96	several, including[oB09]	2012	[ZLQ+12]

the years [KCF+08, LKJ+10, ZLQ+12], though it was already fairly high even when the first techniques were introduced [BTS06, CDGS07]. Furthermore, while initial classification works used many different machine learning techniques, ranging from simple Naïve-Bayes to complex SVM approaches, later work [LKJ+10, ZLQ+12] agrees in identifying C4.5 as the best machine learning tool in terms of classification accuracy.

Notice additionally that (i) results in [ZLQ+12] are gathered over the dataset we publicly released [oB09], and (ii) in the following, we use an improved version of our original proposal [CDGS07], which we fed to a C4.5 classification engine: in reason of the above discussion it is reasonable to expect accuracy results in line with those presented in Table 5.3.

We train classifiers with 8 traffic classes, namely {Web, Encrypted, DNS, Chat, Mail, Network Operation, P2P, Attacks}, additionally to "Unknown" traffic. Note that the focus of this analysis is not the classification accuracy, which has been widely studied by the research community [NA08, KCF+08, LKJ+10], but the computational performance. That is, our focus in this section is the feasibility of these machine learning techniques for on-line classification of network traffic at 10 Gb/s rates.

To gather classification complexity performance, we proceed as follows. We perform a set of experiments by feeding the classification module with classification jobs (i.e., the output of the flow-matching module), configuring our system in the best configuration 1I-2Q-2P. We send synthetic traffic (64B sized packets, five packets per flow) at 14.2 Mpps, which gives a flow rate

of 2.8 Mfps. The experiment span is 60 seconds. Figure 5.10 shows the classification rate obtained for each classifier.

Naïve-Bayes

Classification threads run a custom implementation of Naïve Bayes with Gaussian density estimation. Given the first four packets of a flow have been received, the algorithm associates the flow to the protocol whose model scores the maximum likelihood for the generation of the flow. For each protocol model the algorithm uses the size of the packets as indexes into the four lookup tables that have been associated to that protocol during the training phase: the values extracted are then summed together, we optimize, in fact, the algorithm by storing the logarithms of the table values to avoid products as reported in [RVV+08]. By comparing the values obtained for each of the protocols for which a model is available, the algorithm chooses the application and the classification of the flow terminates: for more details please refer to [CDGS07].

As we can see in Figure 5.10, the Naïve-Bayes module is not a bottleneck and it is able to process every flow produced by the flow-matching module, i.e., up to 2.8 Mfps corresponding to the stressful scenario of 14.2 Mpps. Despite its good performance, Naïve-Bayes is not the best choice in terms of classification accuracy metrics (e.g. false positive and negative ratios) [KCF+08, LKJ+10].

Support Vector Machine (SVM)

As a more accurate alternative we implement a SVM classifier, which was found to have superior precision with respect to Naïve-Bayes [KCF+08]. Figure 5.10 shows that our system is only able to classify $2 \cdot 10^4$ flows per second saturating the two CPU cores dedicated to classification purposes. As 1I-2Q-2P configuration only occupies six cores, we can increment the number of threads up to four, occupying all cores. With this change, the obtained performance increases up to $4 \cdot 10^4$ flows classified per second saturating four CPU cores devoted to classification (i.e., the performance increases linearly

with the number of cores).

Nevertheless, performance of SVM classifier is far from that obtained using Naïve-Bayes ($2.8 \cdot 10^6$). Additionally, as performance increases linearly in the number of cores, this 2-orders of magnitude performance gap cannot be simply filled by throwing more cores.

C4.5 Trees

To fulfill the line-rate requirements while achieving good classification accuracy at the same time, we implement a C4.5 tree classifier, which was found to have the highest classification accuracy by [LKJ+10].

The C4.5 tree obtained after training phase has 185 nodes, of which 93 are leaves, and a depth of 13. As trees place the high-information features at the top of the tree, they limit the tree depth, and so the number of branches. This classifier can be easily encoded in native C using multiple if-then-else branches, which only need to access to four variables – i.e., one for each packet size of the first four ones of a given flow.

As a result, our C4.5 implementation is able to classify all flows in the worst-case scenario of 2.8 Mfps, as it can be observed in Figure 5.10. However, in contrast to Naïve-Bayes-based classifiers, C4.5 trees also offer a good accuracy. In fact, C4.5 performs the best when using the first few packets of flows, because the algorithm discretizes input features during the classification [LKJ+10].

To summarize, we have demonstrated the feasibility of wire-speed statistical traffic classification using state-of-the-art machine learning techniques. In more detail, both Naïve-Bayes and C4.5 trees achieve 2.8 M classifications per second, corresponding to a 14.2 Mpps, while SVM is not suited in this context due to its computational complexity.

5.2.9 Conclusion

We propose an all software solution for statistical traffic classification on commodity hardware. Our system achieves a significant advance with respect to the state-of-the-art in several ways. First, it demonstrates the feasibility

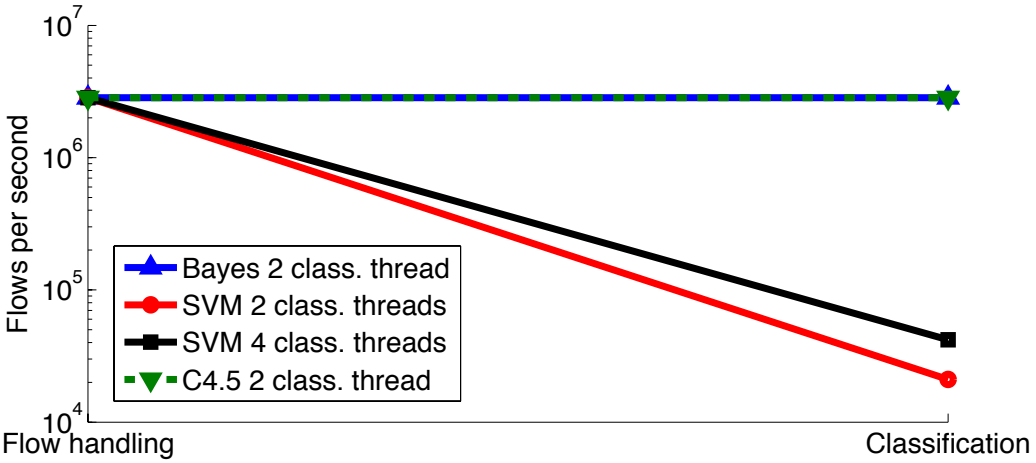


Figure 5.10: Classification performance. Synthetic traffic 64-bytes sized packets

of on-line statistical traffic classification, which was so far confined on off-line analysis published in the literature. Second, it significantly outperforms state-of-the-art classification techniques. Indeed, while the raw classification throughput on real traffic aggregates is about 3× higher than [SGV+10] and 4× higher than [VPI11], however our system is able to sustain flow classification rates 93× higher than [LKJ+10] and 560× higher than [VPI11].

5.3 *DetectPro*: Flexible Passive Traffic Analysis and Anomaly Detection

5.3.1 Introduction

Network monitoring is undoubtedly a key task for network operators due to the ever-increasing users' demand. OTS systems present a more scalable and economic choice than the traditionally used, such as specialized hardware based solutions. The high complexity of current networks in terms of large number of different applications and protocols, aggregates of multi-10Gb/s, tens of million packets per second and millions of concurrent flows per link, requires high-performance along with scalable and flexible designing which

allows processing different network data and granularities (packet-level, flow-level, aggregated statistics) with different purposes (e.g., anomaly detection and traffic classification) simultaneously.

Thus, we have to face the followings challenges:

- *High-performance*: coping with line-rate in 10 GbE links involves processing speeds of up tens of Mpps and Mfps.
- *Scalability*: making it the most of parallelism architectures on common servers and NICs.
- *Flexibility*: monitoring with different tasks simultaneously processing traffic from the same network in different granularities without additional overhead (running on the same PC, processing data without unnecessary copies).

In the literature, we find several capture engines [HJPM10, FD10, Riz12c] that are able to capture all packets received in a 10 GbE link but they do not provide a framework in which easily integrate existing monitoring tools and develop new applications. There are several specific tools [VPI11, GES12, SdRRG+12, JLM+12] which use such capture engines but their purposes are specific and its implementation is monolithic (i.e., it is no possible to simultaneously run with other monitoring applications). On the other hand, the community has proposed and released modular and flexible monitoring platforms, such as tstat[FMM+11], but these approaches, although very useful and versatile, are not able to cope with wire-speed when monitoring 10 Gb/s (and beyond) links.

To overcome all the challenges described above, unlike previous works, we propose a modular architecture system, which is able to obtain and process network traces from different levels and granularities at wire-speed. First, making use of HPCAP capture engine (see Section 3.4.6), the sniffer module is to capture packets from the NIC and copy them to a packet buffer at wire-speed, i.e., more than 14 Mpps in a 10 GbE link. Second, the proposed architecture allows reading packets at user-level with zero copy from different processes and threads simultaneously. Particularly, we have implemented

two modules that generate and collect three essential network traces such as packet-level traces, extended flow-level registers (including, if required, packet payload) and aggregate statistic logs. On the one hand, a packet dumper thread reads packets from the driver kernel-level packet buffer and stores them into disk. On the other hand, a flow manager thread reads packets (concurrently with the packet dumper thread), builds its corresponding flow registers using a hash table and generates fine-grained aggregate statistics (such as throughput in bytes and packets and counting of active flows). A flow exporter thread is implemented to dump flow registers into disk. On upper layers, we can implement different monitoring tasks that simultaneously use the various traces provided by previous modules in a flexible fashion. That is, each task (with its particular purposes and requirements) may be fed with:

(i) Packet-level traces both directly from the driver packet buffer (online) and from the traces generated by the packet dumper thread (offline).

(ii) Flow-level registers both in-memory registers generated by the flow manager module (online) and in-disk registers stored by the flow exporter thread (offline).

(iii) Fine-grained aggregate statistics generated by the flow manager module (offline).

Note that each module may be executed in a different CPU core, thus, taking advantage of parallel processing.

We have evaluated the different modules of the system in several stress test scenarios on a general-purpose server. We checked that the sniffing module is able to receive the maximum packet rate even with small packet size (i.e. up to 14.2 Mpps with 64-byte sized packets). Then, we assessed the different trace generator modules (namely, packet dumper, flow manager and flow exporter) varying different parameters (e.g., packet size, number of packets per flow, number of concurrent flows, IP distribution). We additionally reported the CPU and memory consumption of the proposed system, which illustrates its scalability.

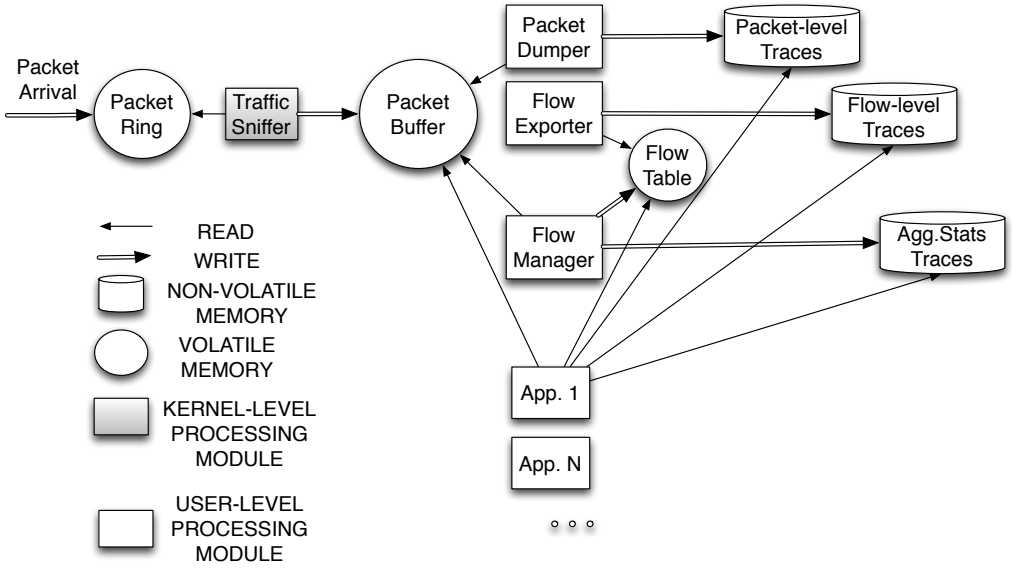


Figure 5.11: DetectPro System Architecture

To show the applicability of our system, we present a network traffic monitoring tool, *DetectPro*, implemented over the proposed architecture, which is able to monitor providing statistics, report alarms and afterwards perform forensic analysis, based on packet-level traces, flow-level registers and aggregate statistic logs. *DetectPro* has been deployed in a commercial network (from a large bank) and its performance evaluation results are presented in this chapter.

5.3.2 System Architecture

Figure 5.11 shows the proposed architecture. The system consists of one kernel-level module, *traffic sniffer*, responsible for capturing packets, three basic user-level modules, *packet dumper*, *flow manager* and *flow exporter*, and a variable number of modules, *application layer*, responsible for specific monitoring tasks which are fed by the output from previous modules.

The workflow in the system is described as follows: each arriving packet is transferred by the NIC via a DMA transaction to a kernel-level packet ring; the packet sniffer module polls the ring for new packets and, if there is

one packet available, copies it in a larger kernel-level packet buffer, accessible from user-level; the packet dumper module reads fix-sized blocks from the packet buffer and writes them into a file in a non-volatile memory device (e.g., a hard disk); the flow manager module reads packets one-by-one from the packet buffer, processes the packet updating a flow table in (volatile) memory and collecting fine-grained statistic logs into disk; the flow exporter module checks the flow table, exporting the expired flow registers (including, if required, the first bytes of payload) into disk; finally, one or more application modules may process one or more previously generated information sources with specific monitoring purposes. Note that the different tasks and modules (sniffing, flow handling, statistics collecting, multiple level traces dumping, specific monitoring) are simultaneously running in different CPU cores, applying CPU affinity (i.e., the execution localization of each thread is fixed on a given core). Consequently, the system makes the most of parallel architectures on multi-core/multi-processors servers.

Traffic Sniffer

This module is devoted to fetch packets from the NIC to a kernel-level buffer, which may be accessed with zero-copy from user-level. A thread running in kernel-level constantly polls for new incoming packets. If a new packet is available, the thread copies the packet to the kernel-level buffer. The module is implemented using HPCAP driver (see Section 3.4.6 for more details).

Multiple listeners in user-level may read concurrently the packet buffer while the kernel-level thread is writing packets in it. This operating mode follows a single-producer/multiple consumer approach. Thus, such architecture allows several applications to process the same traffic simultaneously, exploiting parallelism of multi-core architectures of OTS systems, and consequently, providing a scalable and flexible solution.

Packet Dumper

This module is responsible for generating packet-level traces in disk. Such traces may be offline processed — e.g., with forensic analysis purposes. One

user-level thread implements a packet buffer listener. Such thread reads fix-sized blocks of bytes (e.g., 1 MB) from packet buffer and writes them into disk. To give more manageability and compatibility, packet traces are split in fix-sized files (e.g., 2 GB) and may be trivially converted to PCAP format.

Flow Manager

This module is in charge of two tasks, namely: flow reconstruction and statistic collection. A user-level thread reads packet-by-packet from the packet buffer as a listener. For each packet, its corresponding flow information and the aggregated counters are updated.

To store flows, this module uses a hash-based flow table. The flow table has 2^{24} rows. A hash over the 5-tuple is used as primary key and collisions are handled with lists. Each table entry contains data about the flow, namely: 5-tuple, Media Access Control (MAC) addresses, first/last packet timestamps, counters of bytes and packets, average (as well as standard deviation, minimum and maximum) packet length and interarrival time, TCP statistics (e.g., counters of flags), the first 10 packet lengths and interarrivals and, if required, the first 200 bytes of payload.

A flow is marked as expired when it is not received any packet during a given time interval (e.g., 30 seconds) or when is explicitly finished with FIN/RST flags (only TCP flows). Note that timeout expiration process requires a garbage-collector mechanism. Scanning the whole hash table is not feasible in terms of computational cost. For this reason, we use a list of active flows (each node has a pointer to the flow structure in the hash table). This active flow list is sorted by the last packet timestamp in decreasing order. To expire inactive flows, the garbage-collector only checks the n first active flows (since a flow is checked as active, the rest do not have to be scanned). When a flow is updated with the information of a new packet, the corresponding node in the active list is moved to the end, keeping the list sorted without additional work. Expired flows (for both timeout and flags) are enqueued and will be exported for the next module.

All the memory used (structures, nodes, lists and hash-table) is pre-

allocated in a memory pool to reduce insertion/deletion times, avoiding dynamic allocation/release during the execution. That is, once a data structure is no longer required, the memory is returned to the pool but not deallocated. Then, such data structure can be reused without a new allocation process.

The module periodically (e.g., every second) writes the aggregated statistics into disk. Particularly, generate Multi Router Traffic Grapher (MRTG)-like files with 1-second granularity for three metrics: packets, bytes and active flows. The format of the output file is the following: each line contains a UNIX timestamp and the value of the corresponding counter—packets, bytes or active flows.

Flow Exporter

This module is responsible for exporting flow registers into disk. Such flow-level traces may be offline processed — e.g., with statistical classification purposes. A user-level thread dequeues flows (expired by the previous module) and dumps them to a file in disk. For the sake of manageability and compatibility, flow traces are split in fix-sized files (e.g., 2 GB) and output files may follow the IP Flow Information Export (IPFIX) format [TB11].

To avoid concurrent writing accesses to disk from packet dumper and flow exporter modules (and the resulting potential performance degradation) is advisable to have one independent device for each task.

Application Layer

The system functionality may be flexibly increased with different application modules. The flexibility is obtained in two ways. On the one hand, a simple module may simultaneously use different network data, generated by previous modules, with three different granularities from both live and offline monitoring. For instance, we may implement a traffic classifier that uses both packet-level traces for DPI purposes and flow-level traces for statistical identification. As shown in Figure 5.11, application modules may read:

(i) Packets directly from packet buffer as a listener.

(ii) Packet-level traces in PCAP format from disk.

(iii) Flow information directly in memory.

(iv) IPFIX flow registers from disk.

(v) MRTG-like aggregated statistics in disk.

On the other hand, different modules may be concurrently executed, accessing to data from the same network. For example, we may simultaneously run a NIDSs module and a traffic classifier over the same network traffic. Each module is executed in a different thread. An application launcher creates a thread for running each module routine—as a function pointer in C.

5.3.3 Applicability: a Sample

As an application of the previously proposed system architecture, we have developed a passive network monitoring tool, called *DetectPro*, capable of processing traffic with different aims, such as pattern based anomaly detection, flow-level inspection, traffic trends analysis and selective packet trace collection. Particularly, DetectPro implements several modules in the application layer which are fed with different grained traces—packet-level, flow-level and aggregate.

One module reads aggregate statistics to diagnose both short-term and long-term changes [MGDA12] and report the corresponding alarms to the network manager. If an alarm is triggered, another module reads packet-level traces for subsequent forensic analysis of the anomaly. Several modules are responsible for inspecting flow registers and extract of them information about the network structure—e.g. distribution of host/networks/services with highest flow/packets/bytes counts. Thanks to the flexible architecture of the proposal, new modules may be added, increasing functionality in a scalable fashion. DetectPro has been deployed in a real network from a great bank in Latin America.

5.3.4 Experimental Setup

In this section, we describe the experimental testbed, covering hardware, software and traffic details.

Hardware

Our setup consists of two servers (one receiver and one sender) directly connected with a fiber link. Both servers are based on a dual Intel Xeon E52630 at 2.30GHz with 96 GB of DDR3 SDRAM at 1333 MHz and equipped with a 10 GbE Intel NIC based on 82599 chip. The motherboard model is Supermicro X9DR3-F with two processor sockets and three PCIe 3.0 slots per processor, directly connected to each processor. The NIC is connected to a slot corresponding to the first processor. Regarding the system storage, 12 Serial ATA-III disks conforming a RAID-0 controlled by a LSI Logic MegaRAID SAS 2208 card have been used. These disks are Seagate ES2 Constellation.

Software

Ubuntu 12.04 server 64-bit version is installed with a 3.2.16 Linux kernel. The used filesystem is xfs—preliminary experiments show that is the best choice in terms of performance and scalability, especially when working with large files.

In order to inject traffic, we have developed a tool on top of PacketShader [HJPM10] API, which is able to: (i) generate tunable-size Ethernet packets at maximum speed, and, (ii) replay PCAP traces at variable rates.

Traffic

For our experiments, we used both synthetic traffic and real traces. Synthetic traffic consists of TCP segments encapsulated into 64B-size Ethernet frames, forged with incremental IP addresses and TCP ports.

Real traffic dataset is composed by three packet-level traces. The first trace, called *Backbone*, was sniffed at an OC192 (9953 Mb/s) backbone link of a Tier-1 ISP located between San Jose and Los Angeles (both directions),

Table 5.4: DetectPro performance evaluation datasets

Trace	#Pkts	#Flows	Avg. Pkt Size	Avg. Pkts/Flow
Backbone	570 M	38 M	743	15
BankProxy1	228 M	7 M	661	32
BankProxy2	236 M	7 M	653	33

available from CAIDA [WAcA09]. The average packet size in the trace is 743 Bytes and the average number of packets per flow is 15. The second and third trace, called *BankProxy1* and *BankProxy2*, were captured two different days in November 2012 at a network from a great bank of Latin America. Their traffic corresponds to a proxy from all bank offices and employees—more than 12 K. Table 5.4 shows the characteristics of the traces in detail.

5.3.5 Performance Evaluation Results

In this Section we assess the performance of each module of our system to find its potential bounds.

Traffic Sniffer

Traffic sniffer performance is shown in Section 3.6. Particularly, traffic sniffer is able to receive (using only one reception queue) all packets received for all packet size, except for 60B packet size, when a rate of 12.60 (out of 14.88) Mpps is achieved.

Packet Dumper

First, we evaluate the packet dumper module using synthetic traffic. Figure 5.12 shows the percentage of received packets (during a 60-second experiment) for different packet sizes: $\{60, 64, 128, 256, 512, 1024, 1250,$
$1500\}$. Dumping packet traces does not significantly degrade the performance of traffic sniffer, as long as the disk write throughput is enough. Note that the global write throughput using a RAID-0 is bounded by the write

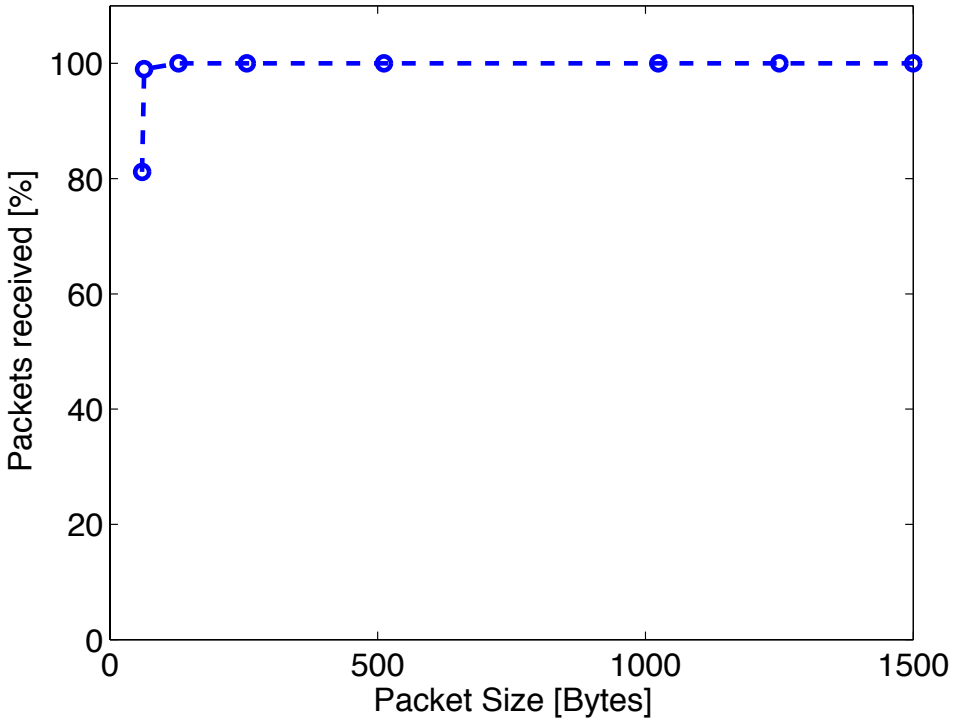


Figure 5.12: Packet dumper module performance. Synthetic traffic

throughput of a single disk multiplied by the number of disks. In our case, as SATA-III disk provides a rate of 150 MB/s, we need, at least, nine disks.

Flow Manager

In order to evaluate the performance of this module, a key metric is the number of concurrent flows rather than the throughput in packets or bytes. Expiration timeout was configured to 30 seconds. We use synthetic traffic and three different real traces with various 5-tuple distributions (Section 5.3.4 for more details). Replaying such three real traces at maximum achievable speed, flow manager is able to process, with zero packet loss, 2.25, 0.38 and 0.42 million concurrent flows, respectively, corresponding to 1.65, 1.85 and 1.86 Mpps.

Flow Exporter

The last experiment evaluates the flow exporter module. Each flow register comprises the information described in Section. 5.3.2. The achievable exportation rate was 82, 46 and 52 Kfps (corresponding to 52, 31 and 35 MB/s of disk throughput) using Backbone, BankProxy1 and BankProxy2 traces, respectively. No packet loss was reported during the experiment.

Computational Resource Consumption

In this section, we evaluate CPU and memory consumption per system module. The results are the average values obtained for a 60-second experiment replaying the Backbone trace. Traffic sniffer occupies about 1 GB of memory (due to the packet buffer) and 75% of one core CPU load (due to its "almost" constantly-polling mode). The memory occupancy of traffic dumper is negligible respect the total of the system, whereas its CPU utilization is near 100%. Flow manager consumes 16 GB of memory, mainly caused by the flow table, and its average CPU load is almost 100%. Finally, the memory consumption of flow exporter module, the same as packet dumper, is almost zero, while the CPU utilization is 90%.

Scalability Analysis

In this section, we aim to assess the system scalability. That is, we evaluate the potential degradation of the system performance due to interaction among different listeners. To this end, we consider a system configuration with one sniffer module and a variable number of listeners—up to the maximum of 11 listeners in our 12-core testbed.

Figure 5.13 shows the percentage of packets received for each configuration. We have used three different datasets, namely: 60-byte and 64-byte packets (synthetic traffic) and the previously described CAIDA trace. The traffic is generated at wire-speed. In the worst-case scenario of 60-byte sized packets, the performance is slightly degraded when the number of queues goes from one to four listeners—linearly from 85% to 80%. The performance remains almost constant when the number of listeners is greater than four. In

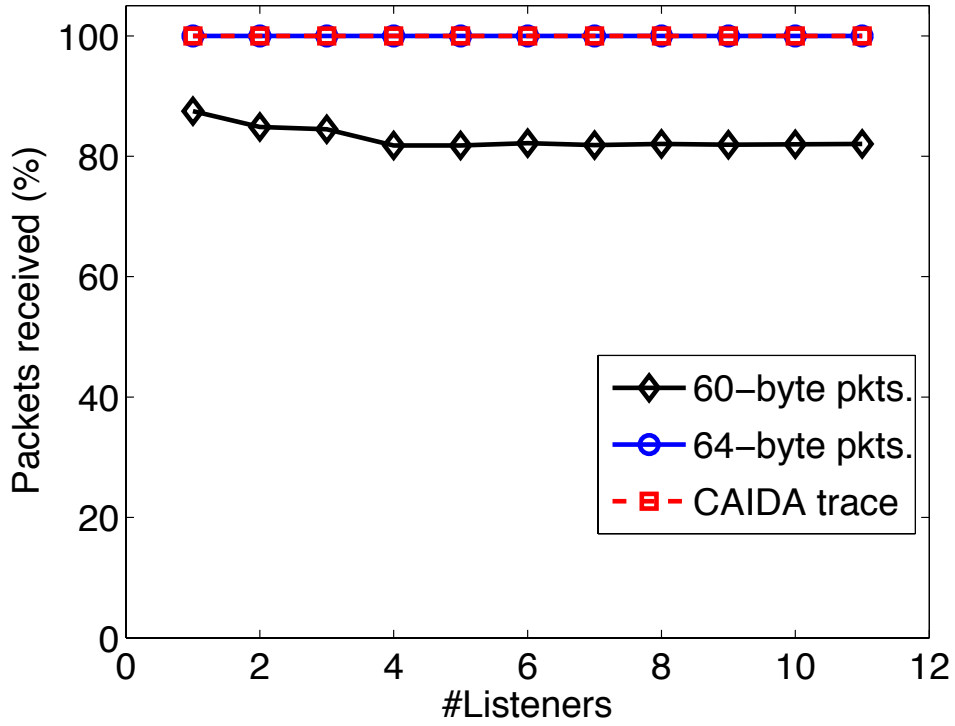


Figure 5.13: Performance evaluation of traffic sniffer according to the number of active listeners

the rest of stressful scenarios (64-byte packets and CAIDA trace), the sniffer module is able to receive all sent packets for all number of listeners.

5.3.6 Conclusion

We propose a flexible and scalable system architecture that is able to monitor network traffic at multi-Gb/s on a general-purpose server. We have evaluated its performance both in a stressing scenario and in several real scenarios (backbone and large commercial network). In addition, we show a real monitoring application, which uses the proposed architecture, which has been successfully deployed in a commercial bank network.

5.4 Summary and Conclusions

Not only high-performance packet sniffing and timestamping is possible on OTS systems, but also upper-level monitoring tasks, such as flow handling and traffic classification, have been proved feasible.

Particularly, we present an open-source statistical classification engine, HPTRAC, able to classify packets at line-rate using commodity hardware. HPTRAC improves the performance of previous works, even in several orders of magnitude. These significant advances with respect to the current state-of-the-art in terms of achieved classification rates are made possible by:

(i) The use of an improved network driver, PacketShader, to efficiently move batches of packets from the NIC to the main CPU.

(ii) The use of lightweight statistical classification techniques exploiting the size of the first few packets of every observed flow.

(iii) A careful tuning of critical parameters of the hardware environment and the software application itself.

Note that the above performance gap between HPTRAC and previous works is hard to remove. Indeed:

(i) While GPUs in [VPI11] could in principle process 40 Gb/s equivalent of traffic, this is forbidden by a bottleneck in the path from the NIC to the GPU. That is, current approaches force to pass through main memory, and waste processing time, to transfer data between the NIC and the GPU creating a bottleneck — although there are preliminary results which may avoid such limitation, they can be only used with Infiniband technology yet [NVI12].

(ii) The statistical technique is anyway much more lightweight than DPI, so that it would benefit more from a GPU. This gap is intrinsic to the nature of the statistical classification process, that avoid transferring packet payload from the NIC to GPUs unlike DPI. Thus, statistical approaches are unachievable for DPI, even making use of different hardware, such as GPUs.

Furthermore, We thoroughly analyze both classification and flow management modules. On the one hand, the detailed comparison of several state-of-the-art machine learning tools points out that C4.5 trees are the best choice due to:

(i) Their known discriminative power [LKJ+10].

(ii) The fact that they can be very efficiently implemented as if-then-else branches, supporting challenging scenarios such as 2.8 M classifications per second.

On the other hand, our exhaustive study of flow management shows that complex structures such as RB trees for collision management does not payoff, and that tables with a smaller memory footprint and carefully chosen hash functions should be preferred. Overall, we conclude that state-of-the-art software structures (namely, balanced RB tree and the Bob-Jenkins hash), are not enough to manage traffic at 10Gb/s in the worst case scenario of 14.8 Mpps on a single core.

This confirms that the multi-core multi-process approach outlined in this study shall be generalized to other high-performance traffic analysis applications required to operate at line-rate. Indeed, we propose a flexible and scalable architecture to develop traffic monitoring systems able to process multi-granularity source of network data, namely, packet-level, flow-level and aggregate from different threads (with different purposes) simultaneously. To show its applicability, we provide a passive monitoring probe, DetectPro, which detects network anomalies, inspects flow registers, analyzes trends and selectively dump packet-level traces for afterwards forensic analysis. We assess the proposed architecture with both synthetic traffic and real traces (from a backbone link and a large commercial network).

Chapter 6

Multimedia Traffic Monitoring in a Very Demanding Scenario

The last few years have witnessed multimedia applications gaining a tremendous popularity. Particularly, different VoIP solutions are increasingly replacing the old PSTN technology. In this new scenario, there are several challenges to overcome. On the one hand, such monitoring process must be able to track VoIP traffic in high-speed networks, nowadays typically of multi-Gb/s rates. On the other hand, recent government directives require that providers retain certain information from their users' calls with security purposes (lawful interception). This implies a significant investment on infrastructure given that traffic monitoring tasks are very demanding in terms of computational power. In this chapter, we propose two novel systems to fulfill such challenges in the cases of: VoIP over SIP (Section 6.3) and Skype (Section 6.4). Such solutions provide very high performance being able to process traffic on-the-fly at high bitrates and with significant cost reduction using OTS systems. To reduce the computational load of on-line traffic classification in high-speed networks, the research community has proposed mechanisms, such as packet sampling. However, the impact of these mechanisms on traffic classification has been only marginally studied. In this chapter, we address such study focusing on Skype application (Section 6.5). Particularly, we have assessed its performance applying different packet sampling rates and

policies.

6.1 Introduction

VoIP has proven to be a mature technology that provides multiple advantages for both telecom operators and users. On the one hand, it allows providers the convergence of their voice and data services into a single network infrastructure. On the other hand, users typically enjoy lower invoices and extended service offering. VoIP technology was developed more than a decade ago and has received much attention by the research community [KP09]. However, it is in the recent years, when VoIP has begun to gain ground to the old PSTN, which still dominates in the voice traffic market. This increase has been strengthened by multiples examples of successful implementations of large-scale VoIP networks: KT Corporation, formerly Korea Telecom, has 2.1 million VoIP subscribers [CSK11]. As the authors in [BMPR10] show, Fastweb, the Internet provider leader in Italy, which offers VoIP telephony exclusively, as of today its number of customers is larger than 600,000. Similarly, Telefónica, in Spain, offers VoIP services to its corporate clients at reduced prices, while PSTN services are still dominant for residential users. However, it is expected that the number of residential VoIP customers will increase sharply in the near future.

In this exciting scenario, providers face some challenges for the successful deployment of VoIP systems in large-scale environments. First, like any other multimedia service, VoIP requires an exhaustive monitoring of the QoS received by the users as well as their QoE in order to provide the same level of quality than PSTN.

Second, such monitoring system must be able to track VoIP traffic in high-speed networks. Currently and in the near future, there is a need for traffic monitoring at 10 Gb/s, or even faster speeds, given the ever-increasing growth of the data transmission capacity [YZ10]. Then, the system that processes such traffic to evaluate its quality has to be fast enough to cope with this data rate.

As a third challenge, we note that new data-retention directives are being

developed in Europe [SGI08] as well as in the US and other countries. These directives require providers to store certain information of the calls carried out by their clients. Such information includes the identification of caller and callee as well as the call start and end times. Such directives may be more demanding in the future, and even include some details of the content due to national security reasons.

Additionally, the entire VoIP call, and not only some descriptors, may be stored in large-scale databases if proper indexing policies are designed and with the consent of the customer. For example, providers can record users' complaints and by-phone contracts of their clients. Similarly, this is useful to evaluate the quality of subcontracted call centers, which are not under the operator direct control, and are extremely popular nowadays. Furthermore, businesses may be willing to record their calls for the evaluation of customer satisfaction. Note that this is a novel monitoring service that can be offered to the customers. In addition, operators can assess if the perceived QoE of a given user is adequate by effectively contrasting the QoS and QoE once the user conversation has been reconstructed.

In recent years, the usage of Skype application is becoming widespread. It is estimated that Skype has over one half billion of users[1] (over 22 million users logged in simultaneously[2]) and it generated 185 million USD in the third quarter of 2009[1].

The analysis, characterization, classification and detection of Skype traffic are gaining considerable interest in the research community [BMM+07, HB08, DCM10, BMMR09]. On the one hand, regulatory bodies are enforcing operators to intercept communications for security reasons (among which Skype calls). On the other hand, Skype's usage from mobile devices (such as smartphones or netbooks) using 3G or General Packet Radio Service (GPRS) mobile networks, is becoming very popular. For these reasons, operators are willing to detect Skype, either to provide differentiated quality of service or to restrict it or with billing/accounting purposes, depending on the contract.

[1] www.techcrunch.com/2009/10/21/skype-hits-521-million-users-and-185-million-in-quarterly-revenue

[2] skypejournal.com/2010/01/skype-dialtone-22-million-online.html

In any case, the detection of Skype traffic is becoming a very important issue.

As data transmission speeds have increased dramatically in recent years, the traffic classification applications are turning out to be a bottleneck for network monitoring. However, the performance evaluation of traffic classification algorithms in terms of processing time, and more specifically, Skype, have received relative little attention.

In this light, this chapter propose two system, named *RTPTracker* and *Skypeness*, to fulfill all the above introduced challenges in the case of: (i) VoIP over SIP and (ii) Skype, respectively.

As previously shown, the ever-increasing data transmission rates have become traffic classification in an exciting challenge. In multi-10Gb/s networks, very common nowadays, traffic classifiers have to be able to capture and analyze up to several tens of millions of packets per second. In spite of improvements on capture capabilities and efforts to optimize and relieve classification mechanisms of burden [NA08], to date many network monitoring systems only deal with packet sampling data in an attempt to reduce such burden. That is, traffic classification systems are not provided with all the traffic but only a fraction of the packets are taken into account.

The relationship between traffic classification and packet sampling was first pointed out in [CEBRCASP11]. In such work, the monitoring system first sampled at packet level, then generated Netflow records, and finally the records were classified using machine learning (ML) techniques [NA08] (specifically, decision trees). Note that Netflow data records only comprise information about the source and destination IP addresses, port numbers, protocol and counters of bytes and packets. Similarly, the authors in [TVRP12] proposed to use packet-sampled flow records that included a more extensive set of features, e.g., RTT or number of ACKs. Both studies concluded that sampling entails a significant impact on the classification performance, especially, in terms of volume in bytes and packets.

Differently, we assume a monitoring system fed with a sample of the total packets traversing the monitored link—instead of analyzing packet-sampled flows. The advantages are twofold: the accuracy increases, and it is possible to classify on-the-fly. Note that flow-based classifying requires that flows

end before being analyzed. This is unacceptable in VoIP applications where operators have to apply measurements, such as accounting, improve quality or, conversely, blocking if some VoIP applications are not allowed by contract, while the call is in course, and not after its finalization.

Specifically, we turn our interest to Skype classification. Particularly, we have evaluated the impact of sampling on the classification of Skype using *Skypeness* over both synthetic and real traces from public repositories.

The rest of the chapter is organized as follows: First, Section 6.2 presents multimedia traffic fundamentals, providing an overview of VoIP technology over SIP protocol and Skype traffic. Then, we propose two multimedia traffic monitoring systems. On the one hand, Section 6.3 describes a novel VoIP monitoring system, called RTPTracker. Section 6.3.2 details each of the modules comprising *RTPTracker* whereas Section 6.3.4 shows the performance evaluation of this proposal. On the other hand, in Section 6.4 we present a Skype traffic identifier, called Skypenes. Sections 6.4.2 and 6.4.3 explain the design of our proposed Skype detection technique, whereas Sections 6.4.5 and 6.4.6 provide a performance evaluation in terms of accuracy and throughput, respectively. Finally, we evaluate the impact of packet sampling on traffic classification in Section 6.5. Particularly, we assess Skypeness's performance applying different packet sampling rates and policies.

6.2 Multimedia Traffic Fundamentals

6.2.1 VoIP: Network Architecture and Traffic

VoIP is a group of technologies that provide mechanisms for voice transmission over IP networks [Goo02]. VoIP brings together different signaling protocols such as SIP or H.323. Proprietary signaling has been used in the early stages of VoIP, however, the widespread deployment of VoIP demands the interoperability of different VoIP equipment vendors in order to establish end-to-end calls. In this scenario, SIP is emerging as the de facto standard for VoIP signaling. As an example, it is being used by all the main US cable operators that give VoIP services [ZR10]. In fact, we have been provided with

data from a popular VoIP operator that also uses SIP as signaling protocol.

The SIP protocol [RSC+02] uses plain-text messages in combination with SDP [HJ98] messages to establish and negotiate the parameters of the associated RTP streams (*call-ID*, available codecs, dynamic ports, etc.) across different systems. Such RTP streams contain encoded voice as part of its data as well as control information.

Figure 6.1 shows the SIP messages transmitted in a common VoIP call. In the initialization phase, the caller sends an *INVITE* message that is answered by the callee with several responses, finishing with a 200 OK response and an ACK that establishes the call. These messages can also be exchanged with proxies, which can act as either caller or callee. Once the call has been established, caller and callee use the information contained in the SDP received from the other end to send the RTP streams with the negotiated IP addresses, port numbers and codecs. It is worth noting that several *INVITE* messages can be sent during a call (the so called re-INVITEs), as SIP protocol implements expiration timeouts and they must be refreshed to keep the call alive. These messages can also be sent if the call is hold, or SDP parameters are changed. To end a call a BYE message is sent, being answered again with a 200 OK response.

The basic commercial SIP VoIP network architectures are based on three elements: SIP Clients, Session Border Controller (SBC) and Media Gateways. SIP clients initiate calls by sending messages to SBCs. Usually, these clients are located behind a multimedia modem/router that separates VoIP traffic from normal traffic. This division is often performed by operators using different VLANs to simplify traffic switching and routing across the access network. SBCs are in charge of switching the signaling and media streams present in SIP communications. Also SBCs act as SIP proxies redirecting SIP calls either to the destination client or to another SBC. However, if a SIP call must be transmitted to the PSTN network or the mobile network, signaling and media stream are redirected to the Media Gateway. Media Gateways are in charge of interconnecting VoIP networks with traditional telephone and cellular networks by making the appropriate conversions between different coding and transmission techniques. Figure 6.2 shows the basic setup of a

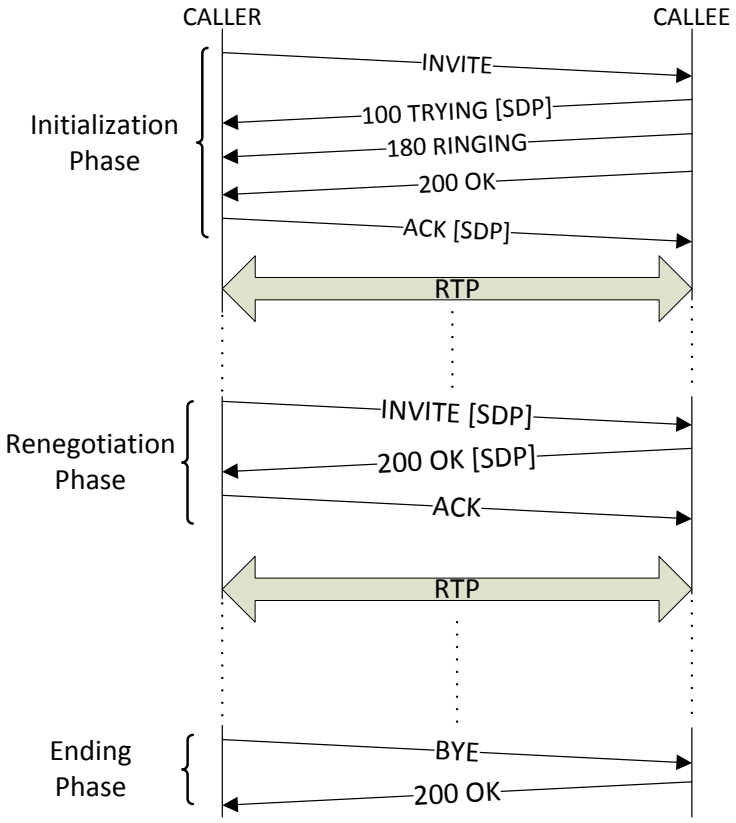


Figure 6.1: Message flow for a typical call: initialization, renegotiation and ending phases

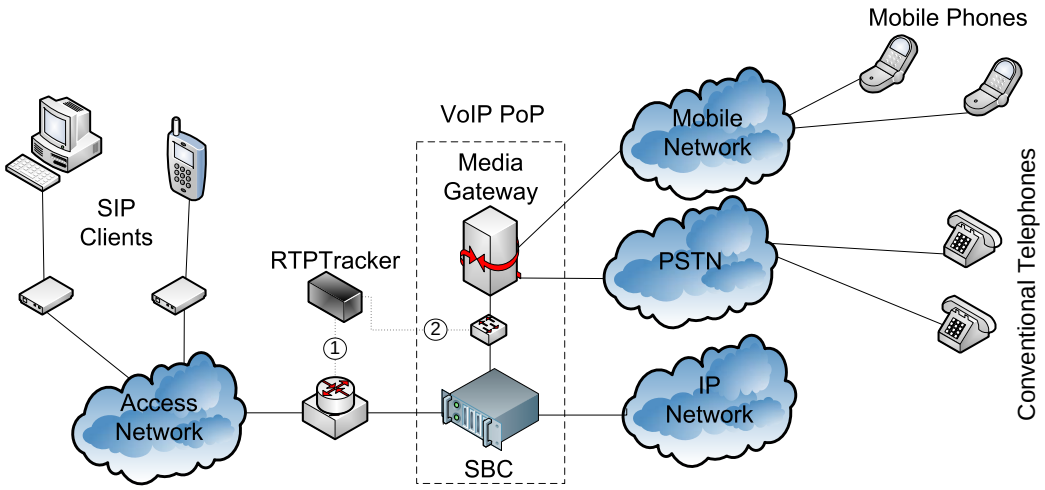


Figure 6.2: SIP VoIP network architecture

commercial SIP VoIP architecture. Usually, this architecture is replicated and spatially distributed to interconnect clients all over the coverage area.

This architecture allows the placement of VoIP monitoring probes in two different locations, either by means of a Switched Port Analyzer (SPAN) port or a tap. The first location is the router between the Access Network and SBC, this provides some advantages as system transparency but increases the capture rate needed as all the traffic is aggregated. The second possible location is the Media Gateway. The advantage of this approach is that the capture rate requirements decrease, because the Media Gateway does no receive cross traffic but end-to-end VoIP calls are not captured.

VoIP makes use of different codecs to transmit voice samples [Goo02]. The most popular VoIP codecs are G.711 and G.729 [KP09]. G.711 provides two different companding algorithms: μ-law (PCMU) and A-law (PCMA), both of them produce data streams at 64 kb/s. G.729 produces lower rate data streams, usually 8 kb/s, but it is based on patents and requires a license to be used. In both cases, the default inter-packet gap is 20 ms, which gives packets of 200 and 60 bytes at the network layer, respectively. Note that the packet size of RTP streams is usually constant and clearly lower than the mean packet length on the Internet, yielding a higher packet rate, which increases the difficulty of capturing this traffic. For example, a full-saturated

10 GbE link (including inter-frame gap, preamble, Ethernet headers and CRC) with G.711 calls gives a rate of 5.3 Mpps whereas G.729 gives a rate of 12.8 Mpps.

The IP fragmentation and TCP segmentation

SIP protocol can be transported either on top of UDP or TCP, involving fragmentation issues when monitoring the calls. In the case of UDP, fragmentation is done at the IP layer. Moreover, when using TCP as transport layer, segmentation is additionally done by the operating system of the SIP message issuer.

The TCP segmentation may cause a SIP message to be split into several segments or even that parts of different messages are transmitted in a single segment (e.g., the end of a message and the beginning of the next one). From the findings of our traces, we have learned that it is very common that an *INVITE* request starts in a segment and ends in the next one. It is also quite usual that response messages such as *100 TRYING*, *180 RINGING*, or *200 OK* are partially found in the same segment. Given that these messages may contain a SDP description, keeping track of the segments and obtaining each message separately are essential actions to correctly identify all the signaling conversations and their associated RTP flows. These cases must be considered since SIP over TCP is an extended choice in commercial networks (in fact, this is the case of our traces): it provides additional reliability to the signaling and allows to identify the end of a connection thanks to TCP flags, which improves the QoE of the VoIP service and reduces keep-alive messages.

We note that the problems that fragmentation/segmentation issue entail by developing high-performance applications using commodity hardware have not been previously analyzed in the literature. This issue undoubtedly deserves the attention of the research community.

There are some other problems, namely, packet loss, duplicated packets and packet reordering. Regarding packet loss, if it is produced during the capture process, lost fragments will not be retransmitted and, as a consequence,

the message will not be reassembled and the call will not be monitored. However, if that fragment is lost by a router or in the network, the sender will issue it again after a timeout implemented by SIP or TCP. This situation will be detected as out-of-order packets. Packet reordering can be resolved using a buffer to recover the monitoring process if a packet or segment is not received when expected. Duplicated packets must be detected to avoid interpreting incorrectly redundant data (e.g., a duplicated *INVITE* message as a second call). In this case, the 5-tuple {source address, source port, destination address, destination port, transport protocol} plus the TCP sequence number can solve the problem. If the SIP messages are transported over UDP, it can be necessary to use the SIP protocol fields (*call-ID* and *CSeq* SIP sequence number) to know if it is a different message or not, because the IP *identifier* field could be set to zero.

6.2.2 Skype Traffic

Traditional services and protocols (such as FTP, web-browsing or SMTP) are not difficult to detect by simple matching to well-known ports. However, such techniques are not enough to detect Skype traffic, which is a proprietary, obfuscated and encrypted protocol that uses per-session random ports. Therefore, not even access to the packet payload is granted and, consequently, well-known DPI [CCR11] approaches are not longer valid. Because of this, the research community has proposed novel approaches based on the use of statistical traffic characteristics (or intrinsic traffic characteristics) and further applying, typically, ML techniques [NA08] to classify.

The authors in [BMM+07] presented a Skype traffic detection algorithm based on two statistical techniques: First, they infer a probability distribution of both packet length and inter-arrival time from audio and video codecs used by Skype. Then, it is checked if the empirical distributions of a given flow fit with the hypothesized ones, using a Bayesian classifier. Second, as Skype traffic is encrypted, it is checked if the payload of a given flow follows a uniform distribution, using Pearson's Chi-Square estimator. The algorithm is implemented as a module of Tstat [FMM+11]. However, Tstat

documentation explains that the Bayesian classifier configuration requires a fine parameter configuration and significant computation load limiting its applicability to multi-10Gb/s networks. The authors in [ACG+12] propose Skype-hunter, a real-time Skype traffic classifier. Skype-hunter is able to detect several Skype traffic classes, such as end-to-end (E2E) and end-to-out (E2O) calls, file transfers and signaling traffic. The detection algorithm is based on statistical characteristics and patterns of the traffic, such as number of exchanged packets and bytes, Start of Message fields, inter-arrival time. The code of Skype-hunter is not free-access but the pseudo-code is included in the article.

The authors in [AZH08, AZH09] use ML techniques (such as AdaBoost, classification trees C4.5 or SVM) to design a simple classifier based on rules, starting from a large traffic trace. This classifier uses simple discriminants such as flow duration, bit rate, packet rate, protocol, as well mean, variance, minimum and maximum observed packet length. In addition, it is worth noting the existence of a US patent [VGBR09], which claims the invention of a system and a method to build Skype traffic models and to further apply them to detect Skype traffic among other network traffic.

On other hand, due to the real-time nature of Skype (and other multimedia services), with the QoS requirements which must be accomplished, it is able to change its behavior to adapt to the changing conditions of the network, namely: bandwidth reduction, delay, jitter or packet loss rate increase. In this light, the authors of [DCMP07] study the congestion control mechanism used by Skype. To this end, an experimental testbed with two computers where the packet input is modified to add delays and shape available bandwidth. The same authors propose a mathematical model for such congestion control mechanism for audio calls in [DCM10]. They conclude that Skype does not implement a congestion control system that reacts to delays but such mechanism is able to respond to packet losses and adapt to the available bandwidth. Finally, in [DCMP08], the authors analyze the bandwidth adjustment for video calls. The authors of [BMMR09] carry out a detailed study of Skype traffic from both audio and video calls. Particularly, they analyze the impact of different network conditions (in terms of

available bandwidth, packet loss probability and end-to-end delay) on Skype traffic. While previous works study how Skype adapts to changes in network conditions (e.g., changing the used codec), they do not evaluate whether Skype detection methods proposed in the literature, are also able to classify Skype traffic when such network changes happen. For instance, when Skype detects that the available bandwidth is smaller, the Skype packet length is reduced. If such bandwidth reduction happens during a call, the statistical characteristics of packet length (minimum, maximum mean and variance) will be modified. As Skype detectors are based on these statistical features, its identification accuracy may be decreased.

It is an undeniable fact that the Internet is becoming a heterogeneous architecture with multiple access technologies—such as home broadband Digital Subscriber Line (DSL) links, fiber to the home, mobile technologies 3G or Worldwide Interoperability for Microwave Access (WiMAX). Because of this, it is interesting to characterize Skype traffic in real environments: wired/wireless, fixed/mobile. In this light, several works [PV07, CSMBMG07, HB08] analyze the behavior of Skype in different scenarios, e.g., Local Area Network (LAN), Wireless Local Area Network (WLAN), Asymmetric Digital Subscriber Line (ADSL), Universal Mobile Telecommunications System (UMTS), from a QoE point of view. Particularly, the authors of [CSMBMG07] study the Skype behavior in different network scenario, evaluating the quality of calls, measured using Perceptual Evaluation of Speech Quality (PESQ), in different connectivity scenarios. They also assess the perceived QoE when (i) using different end terminals: a common PC or a PDA and they analyze the effect of using a public IP address or calling behind a Network Address Translation (NAT) router. They conclude that the quality of calls made by using a PC is above an acceptable minimum quality, whereas, conversely, the quality of mobile calls are unacceptable, due to the worse computational performance of mobile phones.

As previously stated, none of previous works study Skype traffic detection in mobile or wireless environments or analyze the identification accuracy of proposed Skype classifiers in mobile or wireless networks. In [SHR+09] the authors propose a detection method for 3G networks specifically. This mech-

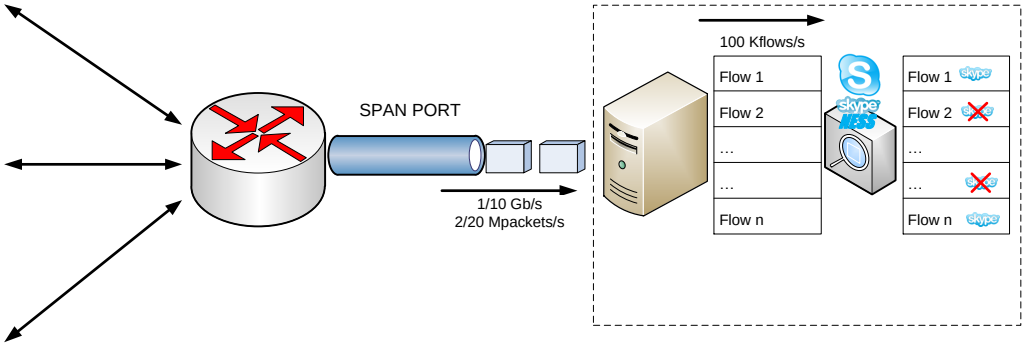


Figure 6.3: Skype traffic identification scenario

anism uses cross layer information available within 3G network as well as packet inspection (such as Skype "header" and payload length). However, such technique is useless with encrypted Skype versions. Therefore, it is more interesting to find out methods based on intrinsic characteristics (e.g., packet length, interarrival times or bit rate) which are able to work with accuracy in different environments both changing network state (e.g., available bandwidth reduction or loss rate increase) and various access scenario—LAN, WLAN, ADSL, 3G.

To the best of our knowledge, there is no state of the art in the performance evaluation of Skype detection from a computational point of view. The authors in [BHA09, DB09, ACG+09] claim that detection of Skype flows is possible with the first 5 or 10 seconds of a given flow, with accuracy greater than 98%. Nevertheless, when a link (or a whole network) is monitored to detect Skype traffic, there are many different flows from other classes of traffic and the authors do not evaluate if their proposed technique can effectively discard non-Skype flows. That is, given a backbone link with many concurrent flows from many different services, the fast classification of Skype flows implies also to discard non-Skype flows at fast rates (see Figure 6.3).

6.3 *RTPTracker*: Line-Rate VoIP Data Retention and Monitoring

6.3.1 Introduction

In this section, we propose a system to fulfill all the above introduced challenges, which we have named *RTPTracker*. The system is composed of four modules, which are in charge of:

(i) Capturing traffic at multi-Gb/s rates.

(ii) Identifying and tracking of VoIP traffic.

(iii) Generating the statistics required to ensure users' QoS as well as in compliance with data retention directives.

(iv) Reconstructing and indexing the VoIP calls to provide novel services based on call recording.

As a distinguishing feature from the best-known vendors' solutions [Cis09], *RTPTracker* faces the deployment of monitoring and data retention of VoIP networks from the point of view of the cost that such tasks imply. That is, in large-scale networks many points of measurements are required which multiplies the infrastructure investment. Some studies [SGI08] have deemed such cost at the range of several billion dollars. Therefore, providers should focus on approaches that maximize their monitoring capacity but, at same time, cut the investment down. While higher processing rates can be achieved with specialized hardware, it typically lacks of the required flexibility to include new traffic statistics or specialized analysis, and the cost is high. Thus, the use of commodity hardware turns out to be an interesting option that combines good performance and limited cost [BDKC10]. For example, the authors in [Ker12] pinpoint the relevance that commodity hardware is gaining in both governments and militaries for their computing needs. *RTPTracker* uses commodity hardware to perform all its tasks. Similarly, another important issue is the cost of the integration of the monitoring system in operational

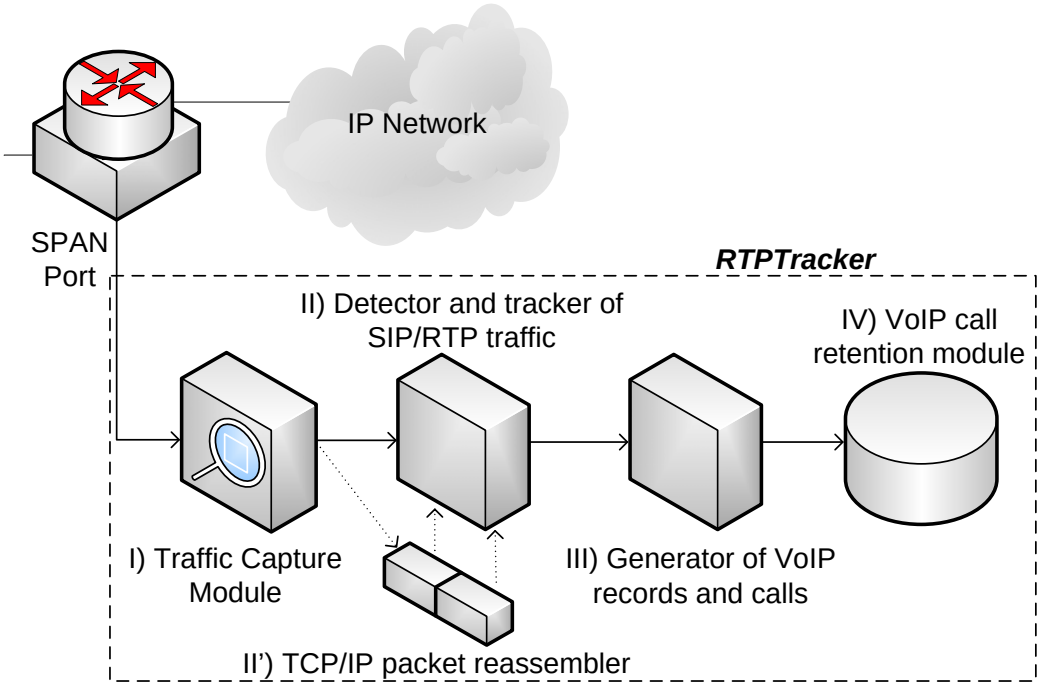


Figure 6.4: RTPTracker architecture

VoIP networks. This calls for monitoring systems whose interference with current VoIP architectures is minimal. This is the case of *RTPTracker*, which only requires a probe fed with the traffic to monitor without any additional network configuration.

6.3.2 System Architecture

In what follows, let us discuss the functionality of each of the *RTPTracker* modules, which are shown and labeled in Figure 6.4. Then, performance evaluation is presented in the next section. *RTPTracker* software is written in C language over a Linux-based system running on a general-purpose server (see Section 6.3.3 for more details about hardware and software setup).

Traffic capture module

RTPTracker requires to keep track and correlate both SIP and RTP data flows but note that these flows do not share the same 5-tuple, and, however,

such tuples are used by RSS technology to distribute packets among different receive queues at NIC and kernel level. Consequently, this prevents the application of multiple receive queues, RSS technique, in SIP VoIP monitoring as a SIP flow and its corresponding RTP flow may potentially end up to different queues and cores. Moreover, the use of RSS produces the appearance of undesirable side effects such as packet reordering [WDC11], which, again, prevents its use. Some novel I/O capture engines achieve 10 Gb/s line-rate packet capture thanks to the application of RSS but, as shown, it is not possible for monitoring VoIP traffic. Additionally, we notice that several proposed capture engines do not perform accurate packet timestamping, which is crucial for monitoring purposes—e.g., to estimate QoS parameters such as jitter. This encouraged us to use HPCAP driver (previously described in Section 3.4.6) that using an only receive queue is capable of achieving 10 Gb/s line-rate packet capture and timestamping.

In this case, the detection module, labeled as II in Figure 6.4), accesses to the driver packet buffer. Essentially, as detailed in Figure 6.5, the driver receives traffic from the NIC by means of DMA transfers and the capture module, running in a kernel-space thread, constantly polls the driver buffer for new incoming packets. If so, the capture module copies the packets to a given memory region that the detection module, is able to map, thus accessing the incoming packets from user-space in a zero-copy basis. The mapped memory region is a circular buffer with two pointers, one for writing (used by the capture module from kernel-space) and another for reading (used by the detection module from user-space). Packets are timestamped when the capture thread copies them to the mapped memory. This is done at kernel-level by means of `getnstimeofday`[3] function, which provides sub-microsecond precision.

While the optimization of I/O packet capture systems has received an extensive study by the research community, the challenges that application developers face by using these new paradigms have only recently arisen expectation. This work translates the principles that those novel I/O packet capture systems apply to improve performance, to the development of a VoIP

[3] `http://man.cx/getnstimeofday(9)`

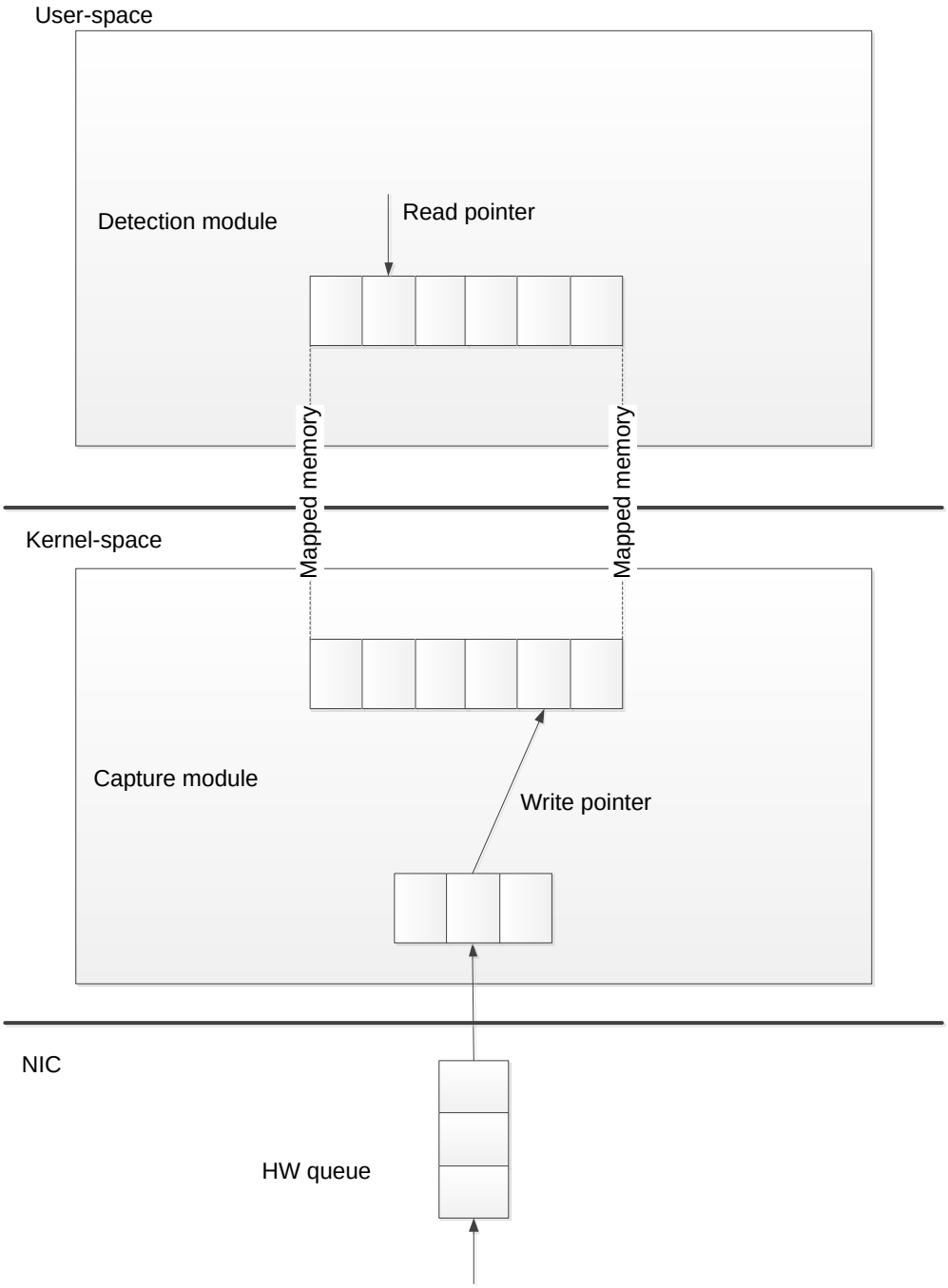


Figure 6.5: RTPTracker capture module scheme: interactions with the NIC and detection module

monitoring system. Specifically, as shown in the following sections, we have imitated memory management mechanisms, i.e., pre-allocation and reuse, and thread and memory affinities.

Detector and tracker of SIP/RTP traffic module

Once a packet is received and timestamped, it can be accessed from this second module, which is responsible for detecting and correlating the SIP and RTP traffic. This second module is executed in a different thread, running at user-space. To achieve peak performance such thread must be tied to a different CPU core than the capture thread of the capture module but in the same NUMA-node. Thus, both threads are running in parallel increasing the power processing and, at the same time, memory affinity is kept reducing memory access latency.

SIP packets contain information of interest to characterize a call such as the call identifier (*call-ID*), caller and callee identifiers (*from* and *to* fields), as well as information to establish the RTP session, essentially source/destination IP addresses and port numbers (4-tuple) and codec of the RTP stream.

Typically, caller and callee exchange the RTP-stream information at the beginning of a new call by means of a couple of SIP packets that comprise in their bodies a SDP message (SIP-SDP) with this information. This step is called initialization phase as was detailed in Section 6.2.1.

However, these SIP-SDP packets can be one of various types of SIP messages (e.g., *INVITE*, *200 OK*, *100 TRYING*, *ACK*) and not necessarily the two first ones. Moreover, RTP 4-tuple can change during the course of a given call, typically named as renegotiation phase (middle of Figure 6.1). This forces to monitor any SIP packet throughout the whole call. Essentially, such a renegotiation is initiated by a new *INVITE* message in a call already in progress. Renegotiations are quite common in SIP. Renegotiation happens periodically, as a mechanism to make sure that connections are still alive, although in such a case, the parameters typically remain the same. Moreover, it also appears when a call is forwarded to another SIP host. This is very common in call centers and offices or due to answering machines.

This second module uses two hash-tables, as shown in Figure 6.6. The first one, (T1), is indexed by SIP *call-ID*, which is useful when a call is updated by a SIP packet; the second one, (T2), is indexed by RTP 4-tuple, which is useful when a RTP packet is processed. Note that these two tables are necessary because RTP packets do not have *call-ID* or other SIP information, whereas SIP packets do not have the same IP addresses/ports as RTP packets. Both hash-tables access to the same data structure by means of pointers. Such structure, (S) in the figure, contains information about the call and its corresponding RTP stream (4-tuple), as well as data about the RTP traffic (that the following module needs to calculate RTP statistics), and the payload of the RTP stream (i.e., call content in raw format).

In the early stages of the system, a VoIP call was marked as expired only when a *BYE* packet is received. However, we observed that a number of calls were not properly closed (e.g., errors in the SIP negotiation phase or malicious traffic) and such calls remained in memory indefinitely. Therefore, it was necessary to implement a garbage-collector mechanism based on timeout expirations. Periodically, the calls in the tables are checked for expiration. The frequency of this expiration considers calls with silence suppression when the VoIP device does not transmit any RTP packet for a while. However, scanning the whole hash-tables requires prohibitive computational costs. Instead, we use a list of active calls, (L), which is scanned to expire inactive (or closed) calls/RTP streams. While the hash-tables are useful to insert and update a call/RTP stream, the active call list is useful to remove them. The nodes of this list have pointers to the structures they represent, and in turn, such structures also comprise pointers to the corresponding SIP and RTP entries in the hash-tables, see Figure 6.6. This makes it possible to remove all the references of a call in both hash-tables by means of the active call list.

Such list is sorted in decreasing order by the last packet arrival time. Thus, the garbage collector only examines the n first active calls up to the first one that does not have to be expired given a threshold, the rest of the calls do not have to be scanned. When a call is updated by accessing to one of the hash-tables, the corresponding node in the active list is moved to the end, keeping the list sorted without any additional work.

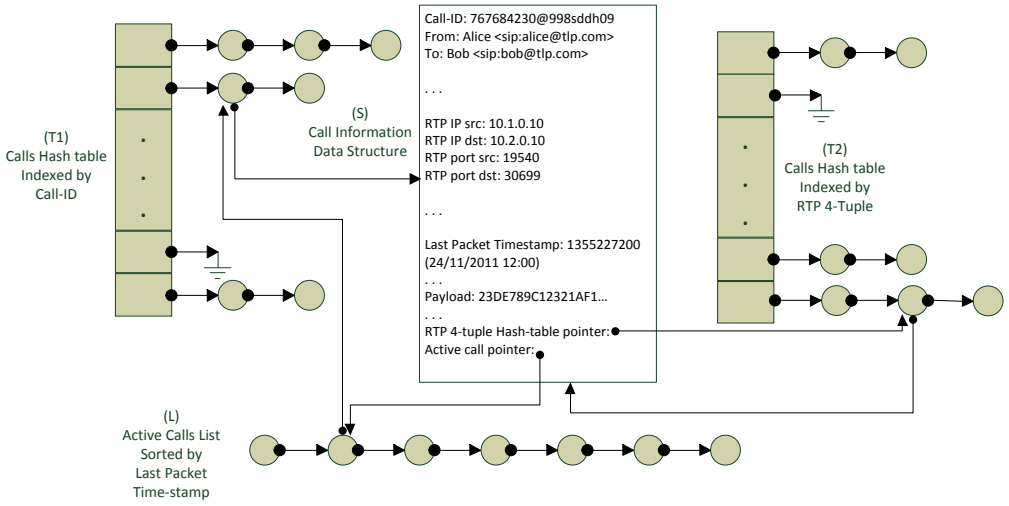


Figure 6.6: Data structures to store and handle calls information in the SIP/RTP traffic detection module

Finally, all the memory used in the application, structures, nodes, lists and hash-tables are pre-allocated in a memory pool to reduce insertion/deletion times, avoiding dynamic allocation/release during the execution. That is, once a data structure is no longer required, the memory is returned to the pool but not de-allocated. Then, such data structure can be reused without a new allocation process. This action has resulted in a significant reduction of the system computational cost.

Bearing in mind all these implementation details, let us summarize the execution flow of this module according to the last received packet:

(i) SIP packets: These packets are easily identified because their source-/destination port is the default port number (e.g., 5060), or other configured port (e.g., 5065, 5070). Then, to classify SIP messages among their different types, we use DPI methodologies. The *INVITE* message is searched as an indication for a new call placement in the active calls table or a renegotiation. In the former case, a new call is inserted in the call hash-table, using the *call-ID* as key. Any subsequent SIP packet with the same *call-ID* is examined to complete the rest of the

required fields. Essentially the IP addresses and port numbers of the corresponding RTP stream but also any additional SIP field considered useful (e.g., *to/from*). Once the IP addresses and port numbers fields are fulfilled, and the connection confirmed by both sides, i.e., *200 OK* and *ACK* SIP messages with correct numbers of sequence, a new entry in the RTP hash-table is created. Otherwise, the connection is ruled out and the memory released. An *INVITE* message with a *call-ID* already present in the call hash-table may indicate the first step of a renegotiation phase. Once the renegotiation is confirmed by the pertinent SIP packets, again *200 OK* and *ACK* SIP messages, the 4-tuple that characterized a RTP stream in the RTP hash-table is replaced by the new one. This allows tracking the entire call despite changes on its RTP parameters. Finally, a *BYE* message causes the call to be exported, as explained in detail in the next section.

(ii) RTP packets: For each UDP packet arrival with even port numbers (i.e., a candidate to be a RTP packet [SCFJ]), the module checks among the active call list entries if one call has been previously placed with the same IP addresses and port numbers. In such a case, the payload is stored and statistics of the calls are updated.

(iii) Remaining packets are discarded.

TCP/IP packet reassembler

As discussed previously, if SIP traffic is transmitted over TCP, an application layer message (in our case, a SIP packet) may be split into several segments or even several messages could be merged into one TCP segment. In fact, this is the case of the real VoIP traces that we have been provided with. Thus, prior to any analysis, SIP messages must be extracted as a unique body from TCP streams and forwarded to the detection module (module II) as shown in Figure 6.4. We have developed an additional module, named TCP/IP packet reassembler (module II'), to deal with this potential problem. Similarly, this module also copes with the IP fragmentation. To summarize, if

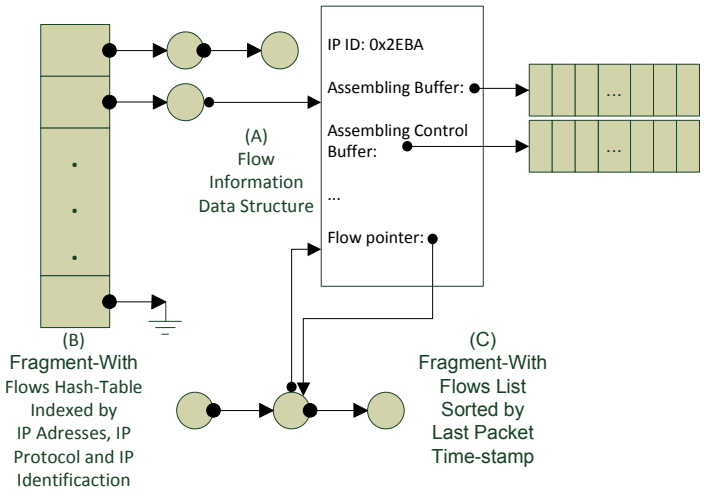


Figure 6.7: IP reassembly data structures

TCP segmentation or IP fragmentation are detected in the network, TCP/IP packet reassembler module must be activated.

The case of IP reassembly is a well-known issue that has already been solved by network stack but whose implementation is not able to deal with high rate speeds. Consequently, we have followed the development principles presented in the previous section. We have used a hash-table to track the fragments of a unique IP packet (indexed by IP addresses and identifier, and layer-4 protocol). Since packet loss may appear, we have again developed a garbage collector as a list of active connections in the same way that the ones explained in the previous section. That is, the hash-table is useful to insert new fragments and the active list is used to rule out those packets that cannot be completed. Figure 6.7 shows these structures. Once all fragments of a given packet have been received, it is forwarded to the following module, either the detection module or the TCP reassembly submodule if necessary.

Similarly, the TCP reassembly submodule uses a hash-table to track connections but, this time, indexes segments by 4-tuple (IP addresses and port numbers). In this case, we aggregate the TCP payloads in order according to their sequence numbers. Then, we deem that a SIP message is completely formed when we have its SIP header and the beginning of a following mes-

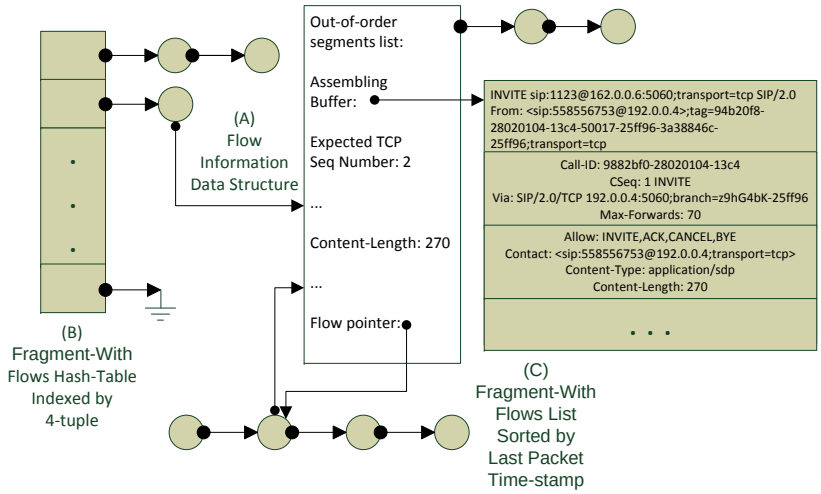


Figure 6.8: TCP reassembly data structures

sage is identified. In this case, we forward the formed SIP message to the detection module. Periodically, the active list is analyzed and those messages that cannot be completed are ruled out. Figure 6.8 depicts these structures.

Generator of VoIP records and calls

When a call expires, it is deleted from the detection module's active call list and hash-tables, and redirected to this third module, which generates all the required information about the call. This includes the information required by the European Union directive 2006/24/EC [SGI08]: the ID of both edges of the communication (*from* and *to* SIP fields), and the start and end times of the call. Furthermore, the module includes several flow parameters in a call record such as the used codecs, count of packets and bytes, throughput, max/min/mean/standard deviation of both packet size and inter-arrival time, round-trip-time, jitter and packet loss rate. It is worth noting that these parameters are useful to assess the users' QoE for a given call. For a further explanation of the calculated parameters the reader is referred to [Sch12].

This module is also able to dump to non-volatile storage the call content in raw format. In addition to the abovementioned parameters, the call record also includes pointers to two file names comprising the packet aggregation of

the RTP communication in both directions. The throughput of the system could be limited in this step by the disk throughput [SWF07]. Several solutions of storage devices can be found in the market, namely: Serial ATA interfaces present a throughput of 1.5, 3 and 6 Gb/s in its versions I, II and III respectively; in turn, the throughput of Serial Attached SCSI is 3, 6 and, expected in a future, 12 Gb/s for its specification SAS-1, SAS-2 and SAS-3. Additionally, the performance of all these storage solutions can be improved using striping techniques, such as Redundant Array of Independent Disks (RAID) 0.

To obtain better disk performance, two parameters shall be configured, RAID strip size and the filesystem request queue size. The former may be fixed using specific tools provided by RAID controller vendors. The latter may be tuned in Linux by means of modifying the system configuration file $/sys/block/ < device > /nr\_requests$: the lower the value contained in this file is, the lower is the amount of petitions stored in the Operating System (OS) queue, thus reducing the use of OS swap memory when writing data into the RAID disk. Specifically, we obtained the best results, as shown in Section 6.3.4, using sizes of 1 MB for the RAID-controller blocks and by minimizing OS cache writings (by fixing $nr\_requests$ to the minimum value, 4, into the mentioned file).

Moreover, it is not necessary to store all the internetwork traffic but only the RTP payloads ignoring headers and cross traffic, which further reduces the required throughput of the storage media. For instance, using G.711 codec, the worst-case scenario, the maximum required rate for call payload storage in a 10 Gb/s link is 6.72 Gb/s (160 RTP payload bytes / 238 total packet bytes $\cdot$ 10 Gb/s).

VoIP call retention module

Periodically, the output of the previous module, i.e., calls records and pointers to the RTP files (if such service is required) are dumped to a MySQL database. The use of a database allows retrieving a call using a given key (e.g., the caller/callee ID). As it turns out, the database size may be too

large to achieve acceptable search times. Thus, the database can be split into several independent databases that can be accessed in parallel. We performed extensive testing of the freeware database MySQL and concluded that a database per day suffices to have a satisfactory search time (as shown in the next section), while avoiding excessive fragmentation of the data set in many databases.

If a system user is willing to listen a given conversation, the raw file must be converted to an audible format, e.g., WAV format. To achieve this goal, a script is created which uses open source tools such as sox [SoX13] and Asterisk [Dig13] to convert from the two files that comprise unidirectional raw RTP streams to a WAV file comprising both call directions. This script works together with the SQL database to query and convert calls on-demand.

6.3.3 Experimental Setup

The proposed system is based on an Intel Xeon E5520 with four cores at 2.27 GHz with six modules of 4 GB DDR3 SDRAM equipped with an Intel X520-SR2 10 Gigabit Ethernet NIC which is based on 82599 chipset. Regarding the system storage, 12 Serial ATA-III disks conforming a RAID-0 controlled by a LSI Logic MegaRAID SAS 2208 card have been used. These disks are Seagate ES2 Constellation. On software side, the experiments were performed over a Ubuntu 10.04 running a Linux kernel 2.6.32.

To evaluate the performance bounds of the modules II, II' and III, we have deployed the following controlled scenario. It comprises 12 computers equipped with 1 Gb/s NICs generating VoIP calls at maximum rates by means of Sipp [GJ13]. We note that Sipp application was not developed to achieve a rate of 1 Gb/s, therefore we needed to aggregate traffic from several machines as shown in Figure 6.9 to generate multi-Gb/s traffic. Sipp allows us to generate VoIP traffic according to several traffic models and codecs. More specifically, it allows modeling the call holding time with an extensive set of distributions as well as the inter-arrivals time between two consecutive calls. We have modeled the call arrival process by a Poisson process as shown in [CSK11], and the call hold time with an inverse Gaussian distribution with

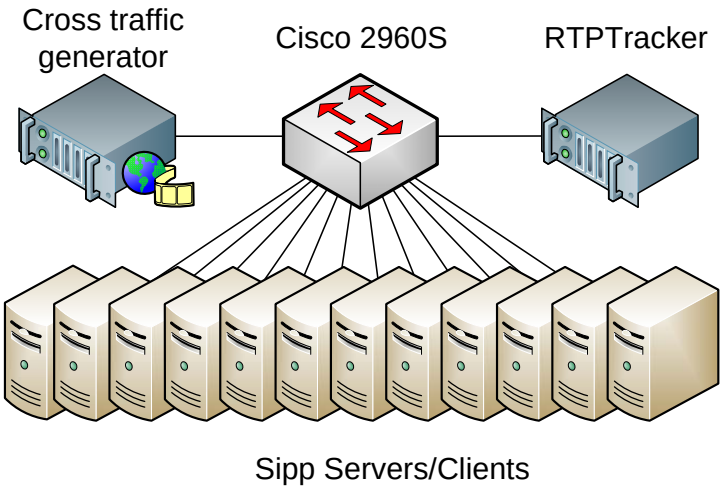


Figure 6.9: Testbed topology

parameters $\mu = 117$ and $\lambda = 19$ as proposed at [BMPR10]. Regarding the codec, we have chosen G.711. In addition, we have used another machine equipped with a 10 Gigabit Ethernet NIC to generate cross traffic. All this traffic is aggregated by a 10 Gb/s switch, Cisco Catalyst 2960S, and then, forwarded by its SPAN port to *RTPTracker* system. With this experimental setup, we have generated VoIP traffic at 8.9 Gb/s, and the rest of the link capacity with cross traffic.

6.3.4 Performance Evaluation: a Stressing Scenario

In this section, we assess the performance bounds of the proposed system, *RTPTracker*, i.e., at what rates the system can work without dropping packets.

First of all, let us evaluate the performance of the traffic capture module. Note that this module captures traffic regardless whether it is VoIP traffic or not and its evaluation is valid for any other purpose. We have performed the worst case scenario experiment, that is, synthetic traffic composed by TCP packets with random payload and layer-2 size of 64 bytes at maximum rate for 30 minutes, i.e., the capture module received more than 1.48 TB of data at 14.2 Mpps, this gives roughly 25 billion packets. In this challenging scenario, no packet loss was reported.

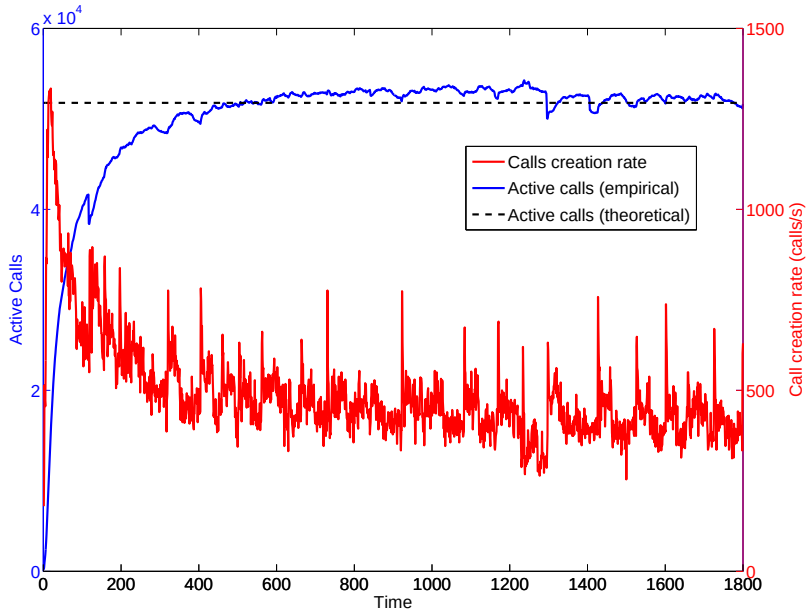


Figure 6.10: Active calls and new calls managed by *RTPTracker* during a 30-minute controlled experiment

Figure 6.10 shows the number of active calls during a 30-minute experiment in this scenario as well as the call generation rate. The number of active calls gives more than 52,000 concurrent calls with a rate of 442 new calls per second in the stationary state, which yields the abovementioned VoIP rate of 8.9 Gb/s.

Attending to storage, at least 6.72 Gb/s of disk write rate must be reached for a full-saturated link with G.711-codec calls (RTP traffic without headers, as detailed in Section 6.3.2). In the abovementioned scenario, this minimum rate should be 5.98 Gb/s. Codec G.711 represents the worst-case scenario because the ratio between the RTP payload and packet size is the highest. Using our experimental setup based on a 12 disk RAID-0 the maximum achieved packet capture and write rate was 9.44 Gb/s, which exceeds the required rate. This figure were obtained after configuring the RAID-controller-cache blocks to have 1 MB and disabling the kernel's cache that operating systems use when applications write in disk. No packet was lost throughout the

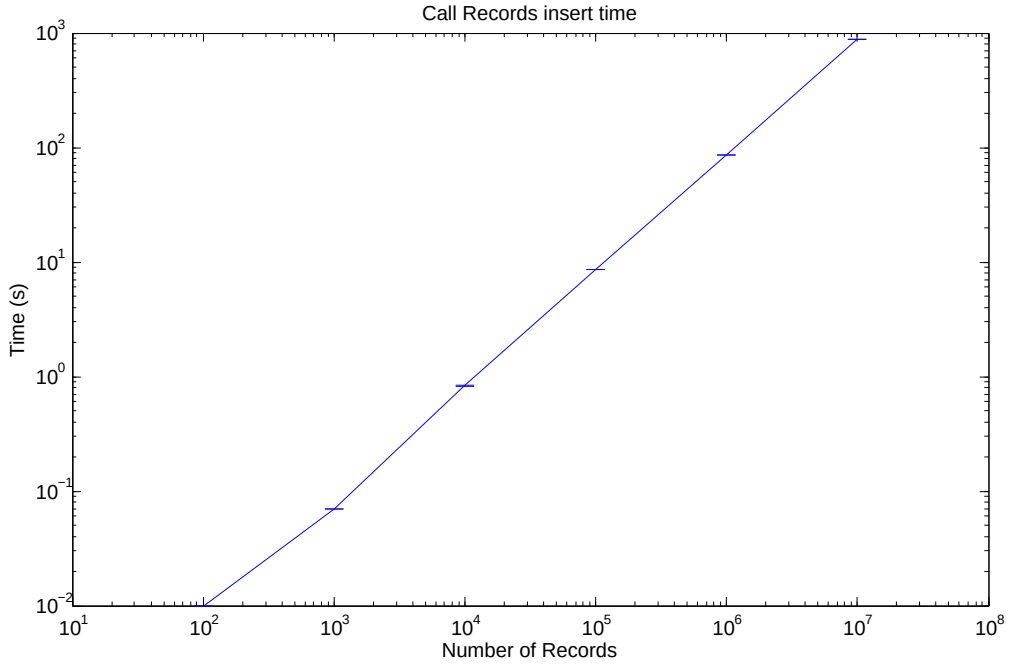
experiment. This supports the high performance capacity of the proposed system, its validity to monitor large-scale VoIP network, and even suggests that its bounds are beyond a 10 Gb/s link.

To quantify the impact of call insertion and the subsequent retrieval in the MySQL database, that is module IV, we conducted the following experiments. The first experiment measures the time devoted to insert call records. The number of records varies between 100 and 10,000,000. Figure 6.11(a) shows a linear increment of the time as the number of records grows. The second experiment measures the time needed to retrieve one record in a database with a number of records varying between 100 and 10,000,000. The SQL query has been built as the worst-case scenario: the statement uses a "*WHERE*" clause involving all the fields of a record. The results in Figure 6.11(b) show a linear increase as the number of records grows. For 10,000,000 records the insert process and the search process take 900 and 6 seconds, respectively. These numbers support the applicability of this approach.

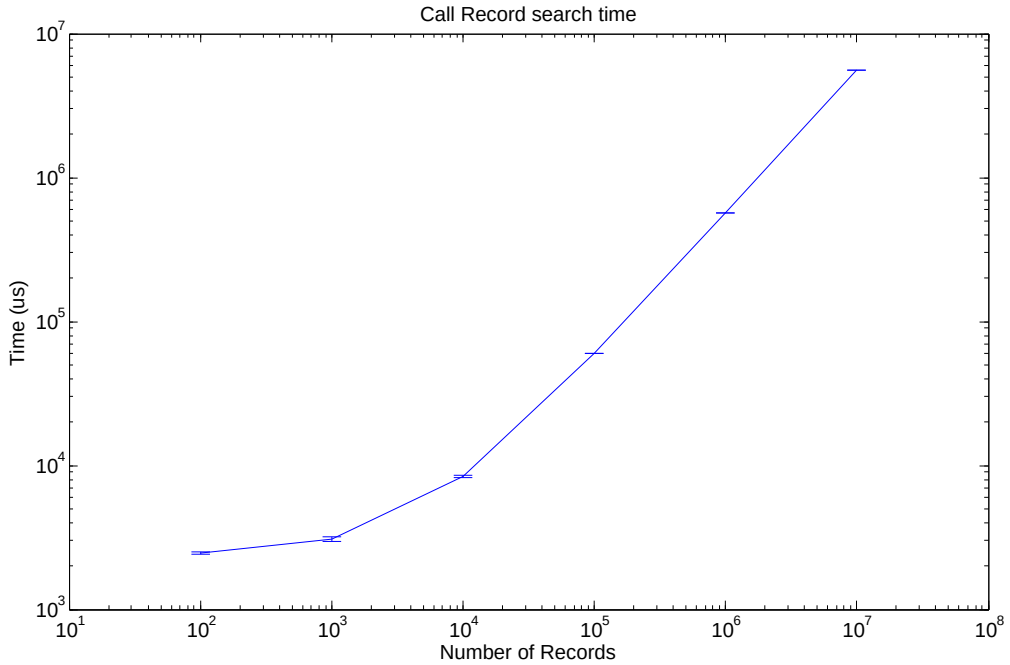
Note that we have two different sets of data, on the one hand the call records, on the other hand the raw RTP streams. For the former, each call record has variable length because the length of SIP caller and callee IDs are also variable (e.g., a phone number or an email). The call records obtained from real traffic traces comprising the fields previously explained show an average call record length of 510 bytes. The average database size obtained for 10,000,000 call records is 4.2 GB.

In the case of tracking RTP raw data, the storage becomes a significant challenge. For the case of an average call of 117 seconds using G.711, 8 kB must be stored per call and direction every second, which gives 1.87 MB of disk space per call. Assuming 10,000,000 calls per day, for full data retention near 18 TB must be stored. This approximation shows the worst-case scenario. Conversely, the most optimistic scenario gives near 2.25 TB per day using G.729.

Finally, Table 6.1 shows the conversion rates of a real trace provided by a VoIP operator, into WAV files. It can be observed that G.711 codec calls are converted faster than the ones from G.729 codec. In the three cases, the



(a) Insert Time



(b) Search Time

Figure 6.11: RTPTracker insert and search times according to the number of records

Table 6.1: Empirical conversion rates from raw to WAV format for G.711 (PCMU and PCMA) and G.729 codecs (the mean call duration is 120 seconds)

Codec	Conversion rate (kB/s)	Conversion rate (calls/s)
PCMU	13,149	6.85
PCMA	19,609	10.21
G.729	1,051	4.38

observed rates are not sufficient to support the throughput of the rest of the system (larger than 400 calls/s). Nevertheless, note that it is not necessary to convert all files. The conversion process can be performed on-demand when the conversation is requested.

6.3.5 Case Study

To put these figures into perspective, we present the traffic intensity figures of a VoIP operator. This operator has fifteen VoIP Point of Presence (PoP)s across Spain as those shown in Figure 6.2. Each PoP serves between 60,000 and 100,000 calls during the busy hour whose mean call hold time is about 120 seconds, being the total number of calls per day a number ranging between 0.6 and 1 million calls per PoP. These results in a mean call arrival rate during the busy hour that ranges between 17 and 28 calls per second and between 2,040 and 3,360 simultaneous active calls. Given these specifications, in computational terms, *RTPTracker* is able to monitor and register calls in each PoP of the VoIP operator previously described, and, even more, it would support a growth of the number of users in, at least, one order of magnitude. Similarly, the authors in [BMPR10] explain that Fastweb VoIP network serves about 32,000 calls in a busy period of 30 minutes. This yields a rate of 17 calls per second with a call duration average of 117 seconds and about 2,000 simultaneous active calls, well below the *RTPTracker* capabilities.

As stated before, the deployment cost is a major issue for network oper-

ators. *RTPTracker* is a cost-effective solution to monitor a VoIP large-scale network. Indeed, the hardware cost of the system to run *RTPTracker* is estimated in 3,800€ (RAID controller and server), 1,100€ (NIC) and, initially, 12 hard disks (as the ones used in the testbed) that cost, in total, about 3,000€. Concerning storage at long term, for sure all user call records must be collected but RTP contents must be only stored when full data retention is required. In this light, the system needs a disk capacity of 126 GB to store all call records in one month. To implement full data retention, the first step is to assess the percentage of users whose calls must be stored. Assuming that 1% of the total calls must be recorded, 540 GB per month would be required. In the worst and unlikely scenario in which all calls must be stored, the amount of 54 TB is needed. It is worth noting that such enormous amount of storage capacity is perfectly feasible to achieve with a single 4U storage chassis. Actually, there are chassis that allow aggregating tens of hard disks and connecting to other chassis in a cascaded configuration to make up a unique solution. In terms of monetary cost, assuming a price of 100€ per TB, the cost of the storage can be estimated in 50€ or 5,000€ per month, according to the percentage of stored calls (1% or 100%). We think that storing more than two years worth of data is not necessary, and this constitutes the limit for the storage capacity. Indeed, note that the data retention directives typically require collecting data during a period between six months and two years.

Finally, thanks to the minimal interference of *RTPTracker* with existing VoIP architectures, we consider the integration cost is basically reduced to the hardware cost, which supports the applicability and viability of our system.

6.3.6 Conclusion

This section shows a novel architecture and system to monitor VoIP traffic, called *RTPTracker*. As distinguishing features, *RTPTracker* provides very high performance being able to process VoIP traffic on-the-fly at high bit rates, significant cost reduction using commodity OTS hardware, and mini-

mal interference with operational VoIP networks. The performance evaluation shows that the system copes with the VoIP load of large operators. We further evaluated the system performance at a 10 Gb/s link and no packet loss was reported. This result demonstrates the scalability of OTS solutions for very high data rates.

6.4 *Skypeness*: Multi-Gb/s Skype Traffic Detection

6.4.1 Introduction

As previously stated, the most state of the art has focused on providing *accuracy* only, regardless of the processing power that is required, which may impair the practical applicability of the traffic classification algorithm in a real-world, high-speed environment. This section aims at filling this gap. Specifically, we seek for Skype detection algorithms that are both *accurate* and *fast*. Our analysis is focused on OTS systems, not specialized software. As it turns out, all Skype traffic classifiers found in the literature run in general-purpose servers. Actually, higher processing rates can be achieved with specialized hardware; however, the hardware solution is less flexible to incorporate changes to the algorithm and the cost is significantly higher. In our case, we trade off complexity of the detection algorithm versus responsiveness for real-time detection purposes, all in an OTS system. We should be able to answer some questions, such as "*Which is the limit rate, per CPU core, for detecting Skype traffic? Is a general-purpose server enough for traffic classification in a highly utilized 1 Gb/s link? How about a 10 Gb/s link?*". The answers of these questions are relevant for network operators with large backbone links, which seek for traffic classification at the minimum expense in monitoring equipment.

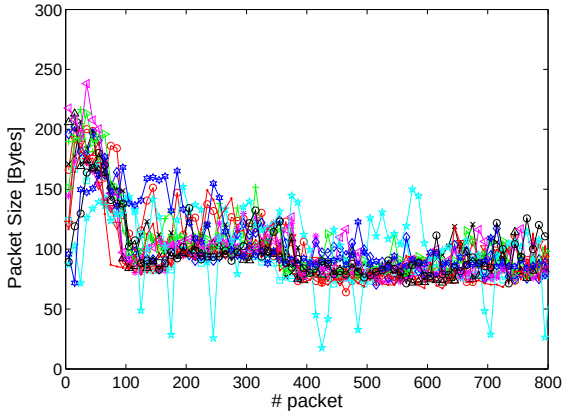
We have implemented a Skype detector, called *Skypeness*. Skypeness is a commodity OTS system to Skype traffic detection at multi-10Gb/s rates based on the functionality of Tstat Skype module [FMM+11] but conve-

niently tuned to satisfy its on-line execution in the current high-speed network requirements. Then, we have evaluated its performance both in terms of accuracy and processing time. Experimental results show that our approach achieves similar results in terms of accuracy than previous work. Additionally, the performance of our proposal, in terms of throughput, shows that Skype traffic can be identified from a traffic aggregate of up to 3.7 Gb/s with a single process, scaling up to 45 Gb/s rates on a four 8-core processors OTS systems.

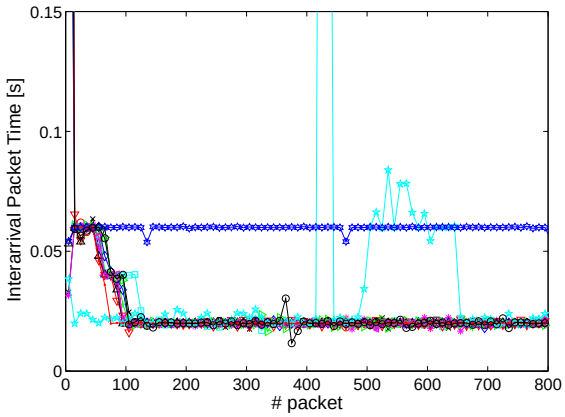
6.4.2 Detector Fundamentals

Skypeness test is based on the statistical techniques presented in [BMM+07]. We do not consider the Chi-Square estimator due to the high-performance requirements (Chi-Square requires inspection of the packet payload). Therefore, our detector uses three intrinsic characteristics of the Skype flows, namely: packet length, interarrival and bitrate. Figure 6.12 shows the behavior of several audio UDP Skype flows in terms of these characteristics: packet length is delimited between 30 and 200 bytes (top), interarrival is nearly constant in multiple of 20 ms (middle) and bit rate is below 100 Kb/s (bottom).

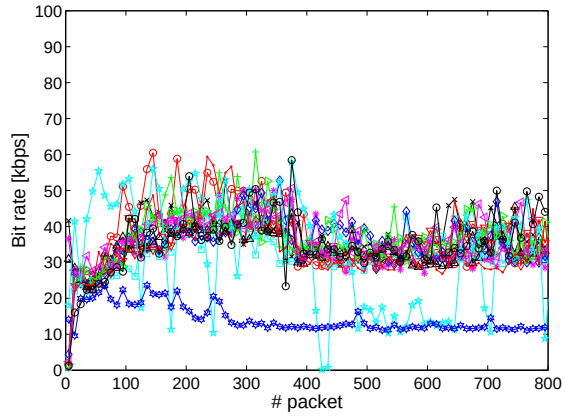
Flow information, such as timestamp (used to compute interrarivals) and size for each packet, is passed to the detector module. In order to smooth data, they are averaged in windows of 10 packets. It is worth noting that we only focus on audio UDP Skype flows with more than 30 packets (3 windows) because this is a validated trade-off threshold to detect Skype calls and ignore control flows. The detector computes the proportion of packet windows whose mean packet size, mean interarrival and mean bitrate are inside the valid intervals. If these proportions are greater than the given thresholds, the flow is classified as Skype. Table 6.2 shows the values for the intervals and the thresholds. The interval values have been chosen with an exhaustive study of Skype flows captured in several scenarios (wired and wireless connection, real and emulated networks conditions). The used dataset will be detailed in Section 6.4.4. The thresholds values have been optimized using



(a) Packet Size



(b) Interarrival



(c) Bitrate

Figure 6.12: Intrinsic Characteristics of a UDP Skype flow (audio conversation)

Table 6.2: Intervals and threshold values used by Skypeness detector

Media	Characteristic	Interval	Threshold
Audio	Packet size [Bytes]	$[60, 200]$	0.75
	Interarrival [ms]	$[i_{n-1} \pm 15]$	0.6
	Bitrate [Kbps]	$[0, 150]$	0.75
Video	Packet size [Bytes]	$[150, 1200]$	0.19
	Interarrival [ms]	$[i_{n-1} \pm 15]$	0.6
File Transfer	Packet size [Bytes]	$[480, 540] \cup$ $[950, 1050] \cup$ $[1310, 1380]$	0.44

C4.5 trees, as in previous works [AZH08, AZH09].

TCP Skype flows are not detected by Skypeness. However, Skype typically uses only UDP as transport-layer because it is more suitable in real-time applications. However, it is uncommon but possible that Skype shifts to TCP in an attempt to evade firewalls or other similar restrictions. As we leverage on packet interarrivals assuming they are fairly constants, and TCP can modify this depending on its configuration, we have focused on UDP traffic.

6.4.3 System Architecture

Skypeness software runs over a general-purpose server based on four AMD Opteron 6128 processors working at 2 GHz. Each processor counts with eight cores and the total memory is composed by 32x4 GB DDR3 memory boards working at 1333 MHz, on a standard Supermicro H8QG6 motherboard. To provide network connectivity one Intel 10 Gigabit CX4 Dual Port Server Adapter is used. This PCIe 16x card uses an 82598EB controller that allows multi-queue transmission and reception up to 16 queues per interface and direction. Figure 6.13 represents the hardware architecture, including PCIe connections.

This server features a NUMA architecture, whereby memory is split into several groups, one per CPU, giving raise to the so-called NUMA nodes (CPU+local memory). Clearly, better performance is achieved when the

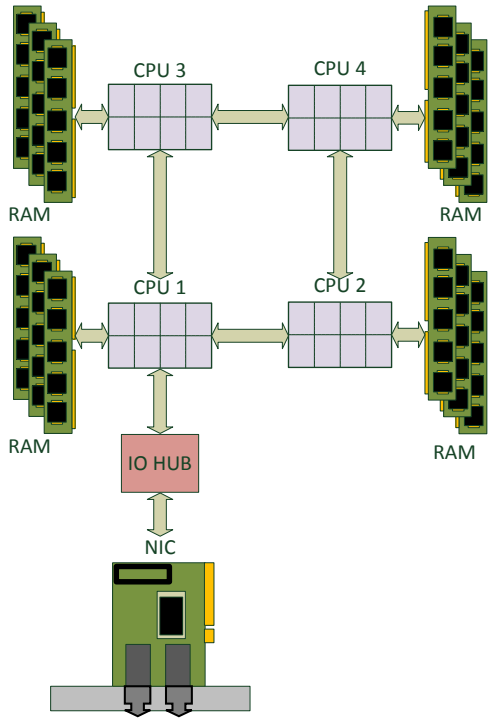


Figure 6.13: Hardware architecture of *Skypeness*

memory region of a process lies within the same NUMA node, such that the CPU executing the process only access local memory. However, the use of memory across different NUMA nodes increases access times and degrades computing performance. Figure 6.14 shows the NUMA design for Skypeness hardware obtained using `hwloc` utility [Pro13]. Following the rationale of better performance when memory locality is exploited, given a NUMA node, a distance vector to other NUMA nodes is defined. The lower the distance is, the higher the performance obtained accessing other NUMA node resources. Table 6.3 shows the NUMA distance matrix for Skypeness hardware obtained using `numactl` utility.

On the software side, Skypeness runs over Ubuntu 10.04 Linux Server (64 bits) using 2.6.35 kernel. Skypeness is divided into three well-distinguished modules. The first module is in charge of capturing and parsing incoming packets. Once a packet is processed, it is redirected to the second module

Table 6.3: Skypeness NUMA nodes distance matrix

NUMA Node	0	1	2	3	4	5	6	7
0	10	16	16	22	16	22	16	22
1	16	10	22	16	22	16	22	16
2	16	22	10	16	16	22	16	22
3	22	16	16	10	22	16	22	16
4	16	22	16	22	10	16	16	22
5	22	16	22	16	16	10	22	16
6	16	22	16	22	16	22	10	16
7	22	16	22	16	22	16	16	10

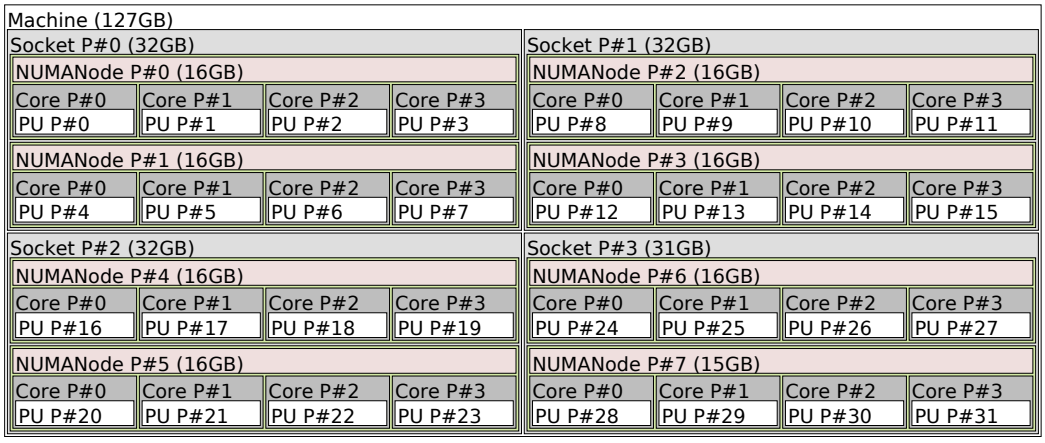


Figure 6.14: NUMA architecture of *Skypeness*

responsible of creating and updating a list of flows or sessions. From now on, the analysis will focus on flows rather than sessions for simplicity but all considerations could be applied as well to sessions. By flow, we mean a stream of IP packets sharing the 5-tuple (IP source and destination addresses, source and destination ports and protocol) and by session, we mean bidirectional flows. This module handles a hash-based flow table in memory to reduce access time to the bare minimum. All the memory used in the application is pre-allocated in a memory pool to reduce insertion/deletion time of a flow in the table.

Periodically, the active flows in the table are checked for expiration. To avoid checking all the active flows in the list, flows are sorted in decreasing order by last packet arrival time. Expiration time-slots happen every 15 seconds because it is the default expiration timeout specified by router manufacturers [Sys12]. When a flow is marked as expired, it is deleted from the active flow list and redirected to the third module that is in charge of running Skype detection tests. If the flow passes all the tests, it is marked as a Skype flow and it is written into a file.

Note that this is a modular architecture that is based on a two-step treatment of flows, first flow extraction, and then, flow classification. In this case, flow classification is Skype versus non-Skype but it could be other. Actually, this software architecture gives flexibility and modularity to the detection tool, making possible the addition of other tests—such as signature based DPI. Figure 6.15 shows Skypeness operation.

6.4.4 Dataset

For the accuracy analysis, we have used three different traces. The first and second traces, named as Trace 1 and Trace 2 in the following, contain Skype traffic captured on the access link of Politecnico di Torino [dT06]. The set of users of such a network are typically students, faculty and administration staff. The measurement campaign duration was 96 hours in May/June 2006. Trace 1 only contains end-to-end Skype voice and video calls whereas Trace 2 only contains Skype out calls. Trace 1 contains ∼40 M packets and Trace

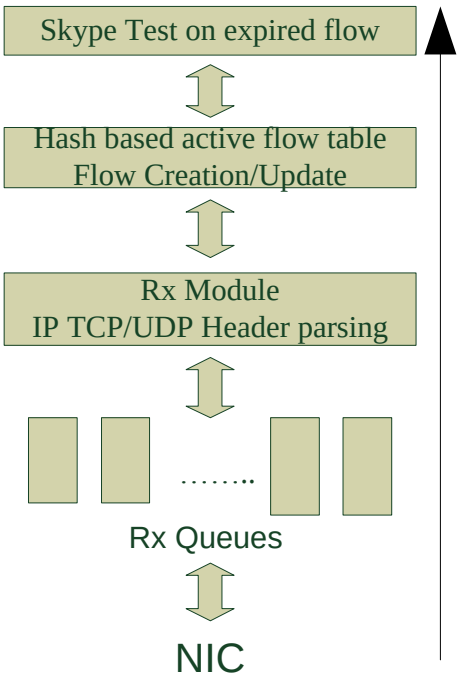


Figure 6.15: *Skypeness* Operation

2 contains ∼3 M packets. The last trace used, named as Trace 3 in the following, is a trace captured in our laboratory at Universidad Autónoma de Madrid. The trace contains ∼22 M packets of P2P traffic from several applications, such as Emule and Bittorrent.

6.4.5 Identification Accuracy Analysis

Table 6.4 shows the false positives/negatives rates in the traces described above. We only consider UDP flows with more than 30 packets, as stated in Section 6.4.2. With traces 1 and 2 we only estimate the false negatives rate (these traces only contain Skype traffic). However, with Trace 3 we estimate the false positives rate (this trace does not contain Skype traffic). It can be observed that the false negative rate is below 1% in Trace 1 and is around

Table 6.4: Skypeness Accuracy Results. (S=Skype, NS=Non-Skype, MP=Million Packets, F=Flows)

Trace		S	NS	Class. S	Class. NS	FP(%)	FN (%)
1	GB	7.81	0	7.77	0.03	-	0.38
	MP	39.46	0	39.15	0.31	-	0.79
	F	1059	0	939	120	-	11.33
2	MB	220.54	0	207.57	12.97	-	5.88
	MP	42.02	0	39.15	2.87	-	6.82
	F	159	0	149	10	-	6.29
3	MB	0	1.05	0	1.05	0	-
	P	0	5312	0	5312	0	-
	F	0	52	0	52	0	-

6% in Trace 2. On the other hand, Trace 3 shows a false negative rate equal to zero.

The obtained accuracy results are similar to the ones found in previous works which use trace 1 and 2: [ACG+09] shows a false negative rate near to 6% (in bytes) in the best case using only statistical classifiers (without inspecting packet payload). In [BMM+07], it is obtained a false negative rate greater than ours, when Naivë-Bayes classifier is used only.

6.4.6 Scalability Analysis: Achieving Multi-10Gb/s Processing Rates

We have connected Skypeness to a server, which reproduces a PCAP file using Tcpreplay [Tcp12]. This tool allows the transmission of pcap traces at variable rate. The transmission rate is varied during the tests (100, 250, 500, 750 and 1000 Mb/s). We have found a limitation in the Tcpreplay throughput to 1 Gb/s (i.e., we have not been able to send the PCAP file faster than 1 Gb/s in spite of using 10 Gb/s NICs).

In this experimental setup, we have only set one reception queue and one traffic classifier instance running in the server. That is, we only use two cores: one for receiving packets and one for detecting Skype flows. Concerning

Table 6.5: Skypeness performance results (per core) in packet, bit and flow rate

Bit Rate [Mb/s]	Packet Rate [Kpps]	Max. Flow Rate per second	Total Packet Loss Rate
100	30	26550	0
250	75	52800	0
500	150	90000	0
750	225	119000	0
1000	300	170000	0

NUMA affinity, we have set the CPU affinity of the reception queue to the NUMA node 1 and the CPU affinity of the Skype detector to the NUMA node 4—the worst case in terms of distance.

For our performance analysis (processing point of view), we have used another trace, named as Trace 4 in the following. Trace 4 was captured from a 3G access network of a Spanish provider. The full trace contains traffic from residential households and small businesses. The trace contains ∼70M packets that correspond to ∼12M TCP/UDP flows captured during ∼18 hours in June 2009.

The results are shown in Table 6.5. For each speed step, we can see the bit rate, the packet rate, the maximum number of flows expired (and consequently analyzed) per second and the packet loss rate in the whole trace. It can be observed that there is no packet loss. It is worth noting that these results have been obtained using only two cores: one for receiving packets and storing them in memory and one for traffic classification. By using the technique proposed in [HJPM10], which assigns a reception queue per socket, we would be able to set up to 16 reception queues and 16 detection processes. In the following, we investigate if the use of this technique would allow performance gains of 16x, which would enable 10 Gb/s Skype traffic classification in a general-purpose server.

To tackle this issue, some offline experiments were made. These experiments use a modified version of Skypeness software that obtains traffic from

a local PCAP trace instead of opening a socket for reception of frames. As it turns out, the theoretical read/write throughput of our DDR3 memory is 170.6 Gb/s, which is by far larger than the bandwidth of an Internet backbone link. To compute the hypothetical bandwidth that Skypeness can handle, the program was executed 10 times and execution times were obtained using Trace 4 as source. This methodology is repeated incrementing the number of parallel instances of Skypeness process and obtaining the corresponding execution times.

Taking into account the NUMA architecture described in Section 6.4.3 the experiments have been designed such that data source and Skypeness software are located on different NUMA nodes and as far, in terms of NUMA distance, as possible. Thus, Trace 4 was located at NUMA nodes 1 and 3 and Skypeness processes were located at NUMA nodes 2 and 4. These conditions set out a worst-case scenario.

Figures 6.16 and 6.17 show the execution time and the throughput versus the number of concurrent instances of Skypeness. It can be observed that the throughput of a single instance is 3.7 Gb/s, scaling linearly up to a remarkable 45 Gb/s classification speed using 16 instances. Note that slope is not 3.7 Gb/s but lower. This is because every NUMA node serializes access to shared memory.

6.4.7 Conclusion

In this section, we develop one example of the use of ML to identify Skype connections. First, some characteristics that Skype traffic shares at the flow-level are first identified. It was found that the packet length, packet inter-arrival times and bitrate permit to extract Skype from the traffic aggregate, as such parameters for Skype traffic vary in relatively narrow ranges. The performance of this proposal, called Skypeness, on a machine with four 2 GHz processors with eight cores each, is far from the capacity of reproducing traffic with the `tcpreplay` [Tcp12] tool used. Therefore, we first evaluated the capacity in a real scenario reproducing 3G traces and obtained that the system was able to deal with `tcpreplay`'s maximum throughput, about 1

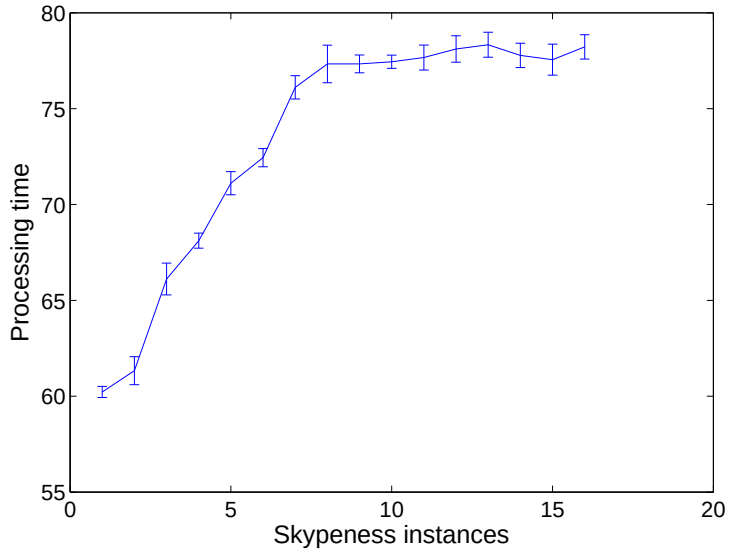


Figure 6.16: Skypeness processing time obtained in offline processing

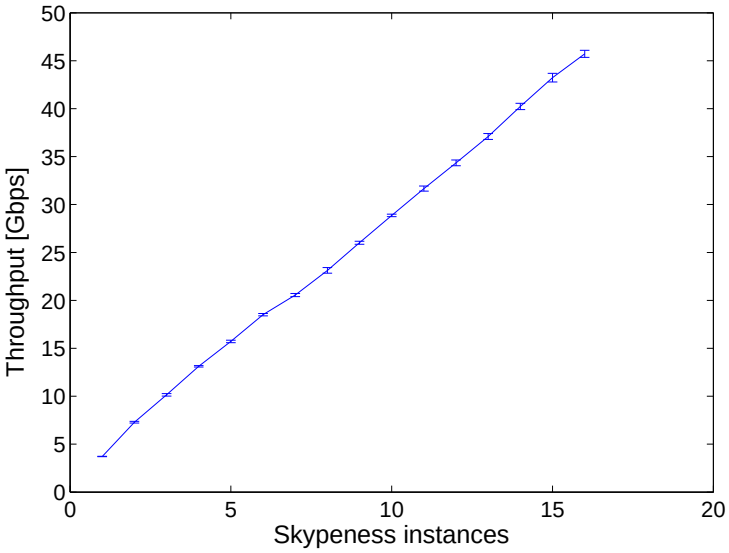


Figure 6.17: Skypeness throughput obtained in offline processing

Gb/s. To evaluate the application at higher rates, we measured the performance after loading the trace in RAM memory. We obtained that a single instance of the software was able to work at 3.7 Gb/s while, by using the 32 cores the system comprised, they achieved a remarkable throughput of 45 Gb/s.

6.5 Packet Sampling Policies: Reducing Computational Complexity

6.5.1 Introduction

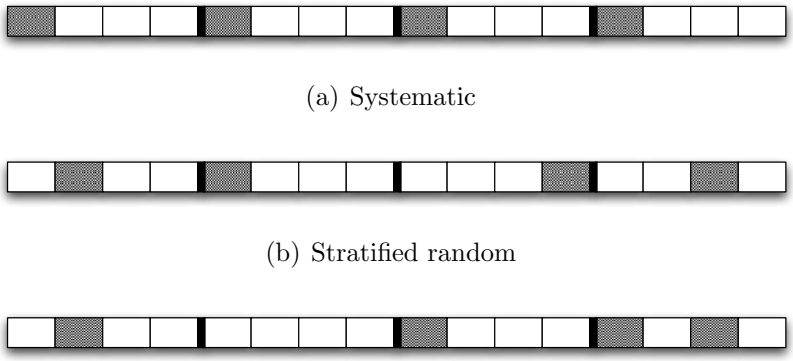
As previously explained, nowadays, traffic classification technology addresses the exciting challenge of dealing with ever-increasing network speeds, which implies more computational load especially when on-line classification is required, but avoiding to reduce classification accuracy. However, while the research community has proposed mechanisms to reduce load, such as packet sampling, the impact of these mechanisms on traffic classification has been only marginally studied.

In this section, we do not analyze packet-sampled flows but assumes a monitoring system fed with a sample of the total packets traversing the monitored link. Thus, we have evaluated the impact of sampling on the classification of Skype using *Skypeness* over both synthetic and real traces from public repositories.

6.5.2 Packet Sampling Policies

Packet sampling techniques allows choosing a fraction of the total amount of packets, following a given criterion to reduce the computational burden of any subsequent analysis. Figure 6.18 shows the three main packet sampling policies [CPB93], namely:

(i) Systematic: data are split in cycles of n packets and the first element of each cycle is deterministically chosen.



(a) Systematic

(b) Stratified random

(c) Simple random

Figure 6.18: Packet sampling policies

(ii) Stratified random: data are also split in cycles of n packets but one element of each cycle is randomly chosen.

(iii) Simple random: each packet is randomly chosen with a given probability $1/n$.

Sampling techniques can be implemented using mechanisms based on either events or timer [CPB93]. That is, each cycle can be either an amount of packets or a time interval. In our case, the cycle is an amount of packets (equal to the inverse of the sampling rate) due to its better performance.

Other packet sampling policies could be applied, such as window-based sampling (i.e., capturing packets during a given period, then, waiting during another time interval without sniffing, and so forth). However, such approaches require capturing all packets (zero losses) in the active period, which is not often suitable in high-speed capturing context.

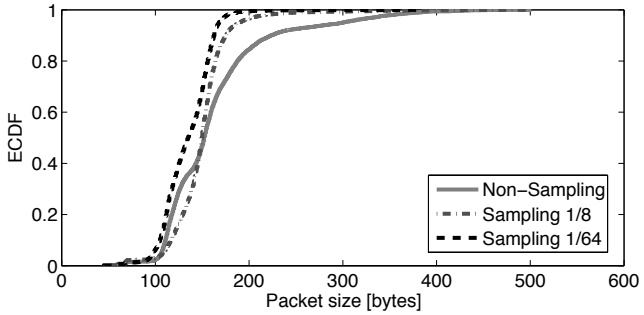
6.5.3 Datasets

We have made use of four different traces of UDP traffic, Table 6.6 shows an overview of the datasets. The first and second traces, named as Trace 1 and Trace 2 in the following, contain Skype traffic captured on the access link of Politecnico di Torino [dT06]. The set of users are students, faculty and administration staff. The capture duration is 96 hours in May/June 2006.

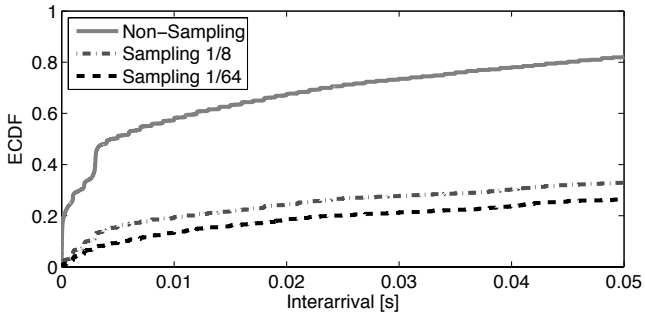
Table 6.6: Datasets to evaluate the impact of packet sampling on Skypeness accuracy

Trace		Skype	Non-Skype	Skype Media
Trace 1	Bytes	8,381,658,970	0	Audio and Video
	Packets	39,458,562	0	
	Flows	1059	0	
Trace 2	Bytes	231,257,652	0	Audio
	Packets	3,049,148	0	
	Flows	159	0	
Trace 3A	Bytes	30,950,000	0	Audio
	Packets	230,100	0	
	Flows	44	0	
Trace 3B	Bytes	108,700,000	0	Video
	Packets	217,300	0	
	Flows	46	0	
Trace 3C	Bytes	162,800,000	0	File transfer
	Packets	254,300	0	
	Flows	46	0	
Trace 4	Bytes	0	1,098,935	-
	Packets	0	5312	
	Flows	0	52	

Trace 1 only contains end-to-end Skype audio and video calls whereas Trace 2 only contains Skype end-to-out calls. Trace 1 and Trace 2 contain 40M and 3M packets respectively. The third trace, named as Trace 3, contains Skype traffic generated in our laboratory at Universidad Autónoma de Madrid in May 2010. The trace contains 700K packets from end-to-end Skype voice (3A) and video (3B) calls, as well as file transfers (3C). The last trace used, named as Trace 4, is a trace generated and captured in our laboratory that contains 5K UDP packets of P2P traffic from several applications, such as eMule and BitTorrent. With this in mind, traces 1, 2 and 3 are useful to estimate accuracy in terms of FN ratio because such traces only contain Skype traffic. TP ratio is estimated with Trace 4 as this trace does not contain Skype traffic.



(a) Packet size



(b) Interarrival time increments

Figure 6.19: Empirical CDF for packet size and interarrival times in audio Skype calls

6.5.4 Performance Evaluation

To assess the effect of packet sampling on the accuracy of Skypeness detector, we have applied the three sampling policies (see Figure 6.18), varying the sampling rate between $1/2^0$ (no sampling) and $1/2^{10}$ over the four packet traces. In the following, accuracy in the case of Skype traces means FN ratio, whereas in the case of Non-Skype trace means FP ratio.

Figures 6.19(a) and 6.19(b) show the empirical Cumulative Distribution Function (CDF) for the packet size and the interarrival time, respectively. Although packet size is not affected by packet sampling (Figure 6.19(a)), interarrival time is distorted when sampling is applied (Figure 6.19(b)) and, therefore, the expected interval values are no longer valid. Thus, Skypeness detection accuracy is reduced to nearly zero in presence of packet sampling.

As an example, Figure 6.20 shows the accuracy of Skypeness (continuous

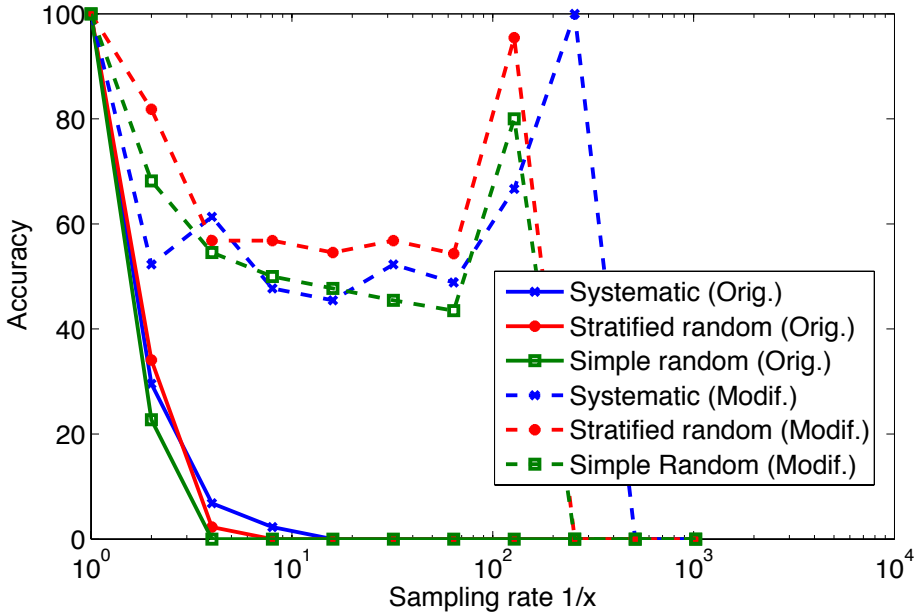


Figure 6.20: Skypeness (original and modified versions) accuracy (in bytes) applying different sampling policies and varying sampling rate over Trace 3A (audio calls)

line) for trace 3A, while Tables 6.7, 6.8 and 6.9 report the results for all traces (roman fonts) applying systematic, stratified random and simple random sampling, respectively. For the sake of simplicity, we only show the results for the cases of sampling rates, $s \in \{1/8, 1/64, 1/128\}$. Note that in the case of Trace 4 $s \in \{1/2, 1/4, 1/8\}$, because there is no enough packets when greater sampling rates are applied (recall that we only consider UDP flows with more than 30 packets).

The accuracy suffers a significant cut even when a sampling rate of only 1/8 is applied for both audio and video traces. This is because mean packet interarrival times do no longer fall inside of the expected intervals assuming unsampled traffic. That is, flows are not identified as Skype calls as packet interarrival time is proportionally incremented with sampling rate, as shown in Figure 6.19(b). Conversely, in the case of trace 3C (file transfer), packet sampling does not have impact on the accuracy because, in this case, the classifier is only based on packet sizes—and packet size distribution is not

affected by packet sampling, as shown in Figure 6.19(a).

In order to adapt the detector to packet sampling, we multiply the observed interarrival times by the sampling rate, thus reducing their values up to the expected intervals when no sampling is applied. Tables 6.7, 6.8 and 6.9 show the accuracy obtained by such modified version of Skypeness detector (italic fonts) applying the different sampling policies. The detector is able to correctly classify, applying systematic or stratified sampling over the Trace 1 (the best case), more than 90% of the traffic regardless the sampling rate. Note that this implies that the detector is able to classify with only 1 out of 128 packets, indeed the results show that the detector after the modification is practically insensitive to the sampling rate. The rest of the traces show also significant accuracy (but the Trace 3A), such accuracy ranges between 73% and more than 95%. In the case of Trace 3A, its accuracy ranges between 54% and 95%, we are investigating on the reasons of this behavior. In Figure 6.20, it is shown the accuracy of such trace in dashed lines. Note that there is a spike in the accuracy when sampling rate is greater than 1/100. This fact may be due to that high sampling rates reduce the number of seen flows removing the more unstable (and difficult to identify) ones. Finally, we observe that the false positive ratio, shown in Trace 4, presents also good results, that is, only a moderate increase.

Thus, this analysis shows that sampling is not a definitive drawback to identify Skype at multi-10Gb/s rates. Particularly, Skypeness would be able to detect Skype traffic at more than 300 Gb/s with notable accuracy, given a sampling rate of 1/8.

6.5.5 Conclusion

We have empirically studied the impact of packet sampling on the open-source Skype traffic detector *Skypeness*, which is based on three statistical features of Skype traffic: delimited packet sizes, nearly constant interarrival times and bounded bitrates. We analyze the effect on the detector accuracy of two packet sampling factors, namely: the sampling rate and the sampling policy.

Table 6.7: Accuracy (% of bytes) of Skypeness detector original version (roman fonts) and modified version (italic fonts) applying systematic sampling

Trace	**Non-Sampling**	**Systematic**		
		1/8	**1/64**	**1/128**
Trace 1	99.59	3.87 *90.72*	0.15 *95.02*	0.04 *95.53*
Trace 2	94.22	35.24 *75.32*	0.54 *85.36*	0.00 *90.86*
Trace 3A	100	2.41 *54.51*	0.00 *56.20*	0.00 *72.19*
Trace 3B	81.38	5.96 *84.48*	0.00 *81.97*	2.68 *82.75*
Trace 3C	95.83	96.24	94.76	96.09
Non-Sampling		**Systematic**		
		1/2	**1/4**	**1/8**
Trace 4	100	100 *83.00*	100 *95.67*	100 *100*

Accuracy dramatically decreases when packet sampling is applied, even with the smallest sampling rates (1/8) due to distortion on the observed interarrival times. We have proposed a simple modification in the detector (to multiply the observed interarrivals by the sampling rate), which lessens the accuracy reduction, at the expense of a moderated increment on the false positive ratio. Thus, this work shows that sampling is not a definitive drawback to identify Skype at multi-10Gb/s rates. Particularly, Skypeness would be able to detect Skype traffic at more than 300 Gb/s with notable accuracy, given a sampling rate of 1/8 and its processing capacity of 45 Gb/s, shown in Section 6.4. Furthermore, the behavior of Skypeness accuracy is very similar when applying different sampling policies. Nevertheless, we can observe that simple random sampling obtains the worst results. Thus, we suggest to use systematic or stratified random as sampling policy.

Table 6.8: Accuracy (% of bytes) of Skypeness detector original version (roman fonts) and modified version (italic fonts) applying stratified random sampling

Trace	Non-Sampling	Stratified Random 1/8	1/64	1/128
Trace 1	99.59	0.61 *90.27*	0.02 *93.55*	1.23 *94.98*
Trace 2	94.22	24.66 *73.85*	0.00 *92.36*	0.00 *96.52*
Trace 3A	100	0.00 *63.02*	0.00 *63.99*	0.00 *94.75*
Trace 3B	81.38	6.05 *86.16*	1.40 *91.51*	10.40 *82.70*
Trace 3C	95.83	96.29	95.29	94.99
	Non-Sampling	**Stratified Random 1/2**	**1/4**	**1/8**
Trace 4	100	98.87 *97.19*	100 *95.53*	100 *79.92*

6.6 Summary and Conclusions

The last few years have witnessed multimedia applications gaining a tremendous popularity. Particularly, a significant sample of this rise is VoIP technology. In this new scenario, there are several challenges for VoIP providers and practitioners. First, VoIP requires a detailed monitoring of the users' QoS and quality of experience QoE to a greater extent than in traditional PSTNs. Second, such monitoring process must be able to track VoIP traffic in high speed networks, nowadays typically of multi-Gb/s rates. Third, recent government directives require that providers retain certain information from their users' calls. Similarly, the convergence of data and voice services allows operator to provide new services such as full data retention, in which users' calls can be recorded for either quality assessment (call-centers, QoE), or security purposes. This implies a significant investment on infrastructure

Table 6.9: Accuracy (% of bytes) of Skypeness detector original version (roman fonts) and modified version (italic fonts) applying simple random sampling

Trace	**Non-Sampling**	**Simple random**		
		1/8	**1/64**	**1/128**
Trace 1	99.59	0.05 *87.65*	11.92 *91.60*	0.17 *93.20*
Trace 2	94.22	9.07 *65.14*	0.00 *71.63*	0.00 *88.92*
Trace 3A	100	0.00 *56.35*	0.00 *58.30*	0.00 *78.49*
Trace 3B	81.38	0.00 *88.73*	0.85 *86.55*	2.76 *70.82*
Trace 3C	95.83	95.98	95.64	96.69
	Non-Sampling	**Simple random**		
		1/2	**1/4**	**1/8**
Trace 4	100	100 *77.26*	100 *68.04*	100 *76.85*

given that traffic monitoring tasks are very demanding in terms of computational power.

On the one hand, this chapter (Section 6.3) has described a practical system to implement full data retention of VoIP traffic on high-speed network scenarios using commodity hardware and the unexpected problems we have encountered by dealing with real VoIP traffic. Our proposal, *RTPTracker*, allows operators to monitor and provide novel services to their customers at the same time that they fulfill current and future data retention directives at reduced cost. Our approach analyzes the traffic on-the-fly to provide timely information to operators and store the content of the calls. The system is designed in such a practical way that minimizes interferences with existing VoIP infrastructure. *RTPTracker* may be connected to a SPAN port of a router traversed by VoIP traffic without any other interaction or configuration, which, again, reduces cost. In addition, *RTPTracker* achieves 10 Gb/s

rate. Such performance has been observed by running experiments on a real implementation and not by simulation. Specifically, our results combine real-world traffic traces along with synthetically generated ones. Such results pave the way for novel services and future scenarios in which, given the increasing popularity of VoIP telephony and other bandwidth-hungry applications, the computational requirements are expected to grow.

On the other hand, in this chapter (Section 6.4), we have proposed an application for Skype traffic classification, named *Skypeness*, that works at 1 Gb/s and 3.7 Gb/s, reading from a NIC and from memory, respectively. In addition, we have assessed such application in our four 8-core processors platform, providing a total throughput of 45 Gb/s. From accuracy point of view, we have obtained a percentage of false negatives of 6% in the worst case whereas the false positive rate is zero, similar to related work. These results show that identification of Skype traffic is feasible at the nowadays high-speed networks, typically ranging from 10 to 40 Gb/s, using commodity OTS hardware.

As previously shown, nowadays, traffic classification technology addresses the exciting challenge of dealing with ever-increasing network speeds, which implies more computational load especially when on-line classification is required, but avoiding to reduce classification accuracy. However, while the research community has proposed mechanisms to reduce load, such as packet sampling, the impact of these mechanisms on traffic classification has been only marginally studied. Finally, in this chapter (Section 6.5), we address such study focusing on Skype application given its tremendous popularity and continuous expansion. Skype, unfortunately, is based on a proprietary design, and typically uses encryption mechanisms, making the study of statistical traffic characteristics and the use of ML techniques the only possible solution. Consequently, we have studied Skypeness, an open-source system that allows detecting Skype at multi-10Gb/s rates applying such statistical principles. We have assessed its performance applying different packet sampling rates and policies concluding that classification accuracy is significantly degraded when packet sampling is applied. Nevertheless, we propose a simple modification in Skypeness that lessens such degradation. This consists

in scaling the measured packet interarrivals used to classify according to the sampling rate, which has resulted in a significant gain. In particular, Skypeness would be able to achieve a classification rate of more than 300 Gb/s with notable accuracy, given a sampling rate of 1/8 and its processing capacity of 45 Gb/s.

Chapter 7

Conclusions

This chapter is devoted to summarize the main results of this Ph.D. thesis (Section 7.1), show the industrial applications where such results are currently utilized (Section 7.2) and outline the envisaged directions for future work and new research lines for continuing the contributions presented in this document (Section 7.3).

7.1 Main Contributions

This thesis addressed the problem of high-performance Internet traffic classification on Off-The-Shelf (OTS) systems. Such solutions are based on commodity hardware and open-source software. We study the feasibility of network monitoring (in particular, traffic classification) at line-rate (10 Gb/s and beyond) using OTS systems. To this end, we analyze each part composing a traffic classification engine, and, in general, a network monitoring system, namely: packet sniffing, packet timestamping, flow matching and statistical classification. Furthermore, we devote a chapter to analyze multimedia traffic due to its relevance and popularity in current networks. The main conclusions from these contributions are presented at the end of their respective chapters in this thesis. However, we outline them in the following list.

(i) **Novel Input/Output (I/O) packet engines proposed in the lit-**

erature are able to capture traffic at multi-Gb/s using OTS systems. Chapter 3 showed that the utilization of commodity hardware in high-performance tasks, previously reserved to specialized hardware, allows packet sniffing at line-rate on OTS solutions. Thus, the performance results exhibited in this chapter, in addition to the inherent flexibility and low cost of the systems based on commodity hardware, make this solution a promising technology at the present. We also highlight that the analysis and development of software based on multi-core hardware is still an open issue. Problems such as the aggregation of related flows, accurate packet timestamping, and packet disordering will for sure receive more attention by the research community in the future. Particularly, packet timestamping issues have been analyzed in Chapter 4 whereas flow matching has been studied in Chapter 5. Other contributions of this chapter are summarized in the following bullet list:

- The limitations of the default networking stack and drivers are identified. Such limitations for packet sniffing of current operating systems waste the potential performance that could be obtained using modern OTS systems (Section 3.2).
- The proposed solutions to circumvent such limitations are detailed. In general, the keys to achieve high performance are efficient memory management, low-level hardware interaction and programming optimization. The adequate tuning of such configuration has proven of paramount importance given its strong impact on the overall performance. Note that this effort of reviewing limitation and bottlenecks and their respective solutions may be also useful for other areas of research and not only for monitoring purposes or packet processing (Section 3.3).
- The different packet capture engines proposed in the literature are reviewed and qualitatively compared. We have identified the solutions that each engine implements as well as their pros and cons (Section 3.4).
- The different packet I/O engines are evaluated in the same OTS

platform to analyze its achievable throughput and scalability. Specifically, we have found that each engine may be more adequate for a different scenario according to the required throughput and availability of processing cores in the system (Section 3.6).

Finally, the contributions in this chapter have led to the following publications (presented in chronological order):

- J. Fullaondo, Pedro M. Santiago del Río, Javier Ramos, J.L. García-Dorado, and Javier Aracil, "AP-CAP framework: Monitorizando a 10 Gb/s en hardware de propósito general", in *Actas de las X Jornadas de Ingeniería Telemática (JITEL),* Santander (Spain), September 2011.

- José L. García-Dorado, Felipe Mata, Javier Ramos, Pedro M. Santiago del Río, Victor Moreno, and Javier Aracil, "Chapter 2: High-Performance Network Monitoring Systems Using Commodity Hardware", accepted for its publication in *Data Traffic Monitoring and Analysis; from measurement, classification and anomaly detection to quality of experience,* Springer, Series in Computer Communications and Networks, editors: C. Callegari, M. Matijasevic, E. Biersack, 2012.

(ii) **Accurate packet timestamping is achievable for high-speed and OTS systems**. Chapter 4 demonstrated that techniques used in novel packet I/O engines for enhancing capture performance, such as batch processing, introduce inaccuracy in packet timestaping. We propose three techniques, based on two different approaches, to mitigate or overcome such inaccuracy. On the one hand, we showed that (uniformly or weightedly) distributing the inter-batch time among the different packets composing a batch increase the accuracy of timestamping. On the other hand, a redesign of the network driver (to constantly poll the Network Interface Card (NIC) buffers for incoming packets and

then timestamp and copy them into a kernel buffer one-by-one), mitigates timestamping degradation. Other contributions of this chapter are summarized in the following bullet list:

- The timestamping inaccuracy, introduced by novel packet capture engines, is over tens of microseconds.
- The timestamping inaccuracy when using our proposals is significantly reduced. Specifically, distributing-time-based methods enhance the accuracy, decreasing the standard deviation of the timestamp error below 200 ns with full-saturated links, whereas polling-based method achieves a standard error of 1 microsecond with a real trace.
- We propose to combine both solutions according to the link load, i.e., when the link is near to be saturated distributing timestamp in groups of packets and, otherwise, using polling-based timestamping packet-by-packet.

Finally, the contributions in this chapter have led to the following publications (presented in chronological order):

- V. Moreno, Pedro M. Santiago del Río, Javier Ramos, Jaime J. Garnica, José L. García-Dorado, "Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines", *IEEE Communications Letters*, **16** (11) (2012), pp. 1888–1891.

(iii) **Wire-speed flexible traffic monitoring and statistical classification are feasible using commodity-hardware-based and software-only solutions**. Chapter 5 showed that on-line traffic classification based on statistical fingerprints is possible thanks to: the use of an improved network driver to efficiently move batches of packets from the NIC to the main Central Processing Unit (CPU); the use of lightweight statistical classification techniques exploiting the size of the first few packets of every observed flow; and a careful tuning of critical parameters of the hardware environment and the software application itself.

Indeed, while the raw classification throughput on real traffic aggregates is about 3× higher than [SGV+10] and 4× higher than [VPI11], however our system is able to sustain flow classification rates 93× higher than [LKJ+10] and 560× higher than [VPI11]. More generally, the proposed system architecture was proved flexible and scalable to develop traffic monitoring systems able to process multi-granularity source of network data, namely, packet-level, flow-level and aggregate, from different threads (with different purposes) simultaneously. Other contributions of this chapter are summarized in the following bullet list:

- When using multi-queue and multi-core capabilities is preferable to avoid contention in the access to shared data structures, such as the hash table and the job ring. To arbitrate concurrent access to shared memory is necessary to perform synchronization and locking operations, which cause a huge performance loss (Section 5.2.5).
- Complex structures such as Red Black (RB) trees for collision management do not payoff, but tables with a smaller memory footprint and carefully chosen hash functions should be preferred. Overall, we conclude that state-of-the-art software structures (namely, balanced RB tree and the Bob-Jenkins hash), are not enough to manage traffic at 10Gb/s in the worst case scenario of 14.8 Million packets per second (Mpps) on a *single* core (Section 5.2.7).
- The detailed comparison of several state-of-the-art machine learning tools points out that C4.5 trees are the best choice due to: its known discriminative power, and the fact that they can be very efficiently implemented as if-then-else branches, supporting challenging scenarios such as 2.8 M classifications per second. Other Machine Learning (ML) tools are not advisable due to: its smaller classification accuracy, e.g., Naïve-Bayes, or its enough throughput, e.g., Support Vector Machine (SVM) (Section 5.2.8).
- We released the code of both the proposed classification engine and the traffic injection engine used in the experimental testbed [Hrg12].

Finally, the contributions in this chapter have led to the following publications (presented in chronological order):

- Pedro M. Santiago del Río, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil, "Wire-Speed Statistical Classification of Network Traffic on Commodity Hardware", in *Proceedings of ACM International Internet Measurement Conference (IMC),* Boston, MA (USA), November 2012.

- Pedro M. Santiago del Río, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil, "Early traffic classification beyond 10 Gbps using commodity hardware", under review in *Computer Communications, Elsevier.*

(iv) **Full-data retention and monitoring of Voice over IP (VoIP) (both over Session Initiation Protocol (SIP) and Skype) traffic at multi-Gb/s rates is manageable by using only OTS systems**. Chapter 6 demonstrated that it is possible to fulfill the challenging requirements entailed by multimedia traffic monitoring in current high-speed networks (10 Gb/s and beyond). We propose two novel modular systems, for the cases of VoIP over SIP (Section 6.3) and Skype (Section 6.4) respectively, which provide very high performance being able to process traffic on-the-fly at high bitrates and with significant cost reduction using OTS systems. Other contributions of this chapter are summarized in the following bullet list:

- It is not advisable to use multiple receive queues when is required to keep track and correlate both SIP and Real-time Transport Protocol (RTP) data flows because SIP flow and its corresponding RTP flow may potentially end up to different queues and cores (Section 6.3.2).
- Two independent tables (one for indexing SIP and the other one for indexing RTP) are necessary to manage VoIP over SIP traffic

because RTP packets do not have *call-ID* or other SIP information, whereas SIP packets do not have the same Internet Protocol (IP) addresses/ports as RTP packets (Section 6.3.2).

- If Transmission Control Protocol (TCP) segmentation or IP fragmentation are detected in the network, a TCP/IP packet reassembler module must be implemented to be able to identify traffic (Section 6.3.2).
- The audio conversion process (from raw to an audible format) must be performed on-demand when the conversation is requested because the conversion rates are not sufficient to support the throughput of the rest of the system in a 10 Gb/s network, larger than 400 calls/s (Section 6.3.4).
- The experimental results show that identification of Skype traffic is feasible at the nowadays high-speed networks, typically ranging from 10 to 40 Gb/s, using commodity OTS hardware. Specifically, Skype traffic classification works at more than 3.5 Gb/s using only one core. This process adequately scales, achieving a throughput of 45 Gb/s using four 8-core OTS processors (Section 6.4).
- Packet sampling impacts on some traffic features, such as interarrival time—while packet size distribution is almost unaltered. Consequently, classification accuracy is significantly degraded when packet sampling is applied (Section 6.5).
- Classification accuracy degradation may be mitigated by scaling the measured packet interarrivals used to classify according to the sampling rate. Thus, Skypeness would be able to achieve a classification rate of more than 300 Gb/s with notable accuracy, given a sampling rate of 1/8 and its processing capacity of 45 Gb/s (Section 6.5).

Finally, the contributions in this chapter have led to the following publications (presented in chronological order):

- Pedro M. Santiago del Río, Javier Ramos, J.L. García-Dorado, Javier

Aracil, Antonio Cuadra-Sánchez, and Mar Cutanda-Rodríguez, "On the processing time for detection of Skype traffic," in *Proceedings of 7th International Wireless Communications and Mobile Computing Conference (IWCMC),* Istanbul (Turkey), July 2011.

- Pedro M. Santiago del Río, Diego Corral, José L. García-Dorado, and Javier Aracil, "On the Impact of Packet Sampling on Skype Traffic Classification," accepted for its publication in *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM),* Ghent (Belgium), May 2013.
- José L. García-Dorado, Pedro M. Santiago del Río, Javier Ramos, David Muelas, Víctor Moreno, Jorge E. López de Vergara, Javier Aracil, "Low-cost and High-performance: the case of VoIP monitoring using commodity hardware", under review in *Journal of Network and Systems Management, Springer.*

7.2 Industrial Applications

The results and applications of this thesis are being currently exploited by Naudit HPCN [CN13]. Naudit is a technology-based startup created as a spin-off from two universities: Universidad Autónoma de Madrid (UAM)[1] and Universidad Pública de Navarra (UPNA), and it is part of its Campus of International Excellence[2]. Its shareholders include both universities as well as Spanish National Research Council (CSIC) by way of Madrid Science Park (Parque Científico de Madrid).

Naudit along with Fundación de la Universidad Autónoma de Madrid (FUAM)[3] have carried out several innovation and technology transfer projects. Among Naudit clients, there are public organisms like Spanish Industry Min-

[1] `http://www.uam.es/ss/Satellite/es/1242657608103/listadoCategorizado/Spin-offs_de_la_UAM.htm`

[2] `http://campusexcelencia.uam-csic.es/`

[3] `www.fuam.es`

istry, telecom operators like Movistar, multinational banking groups like BBVA, industrial companies like Airbus or important energy producers.

Specifically, the followings results of this thesis are directly applied in the industry:

(i) **Traffic Analysis and Anomaly Detection**: The flexible passive monitoring probe (described in Section 5.3), *DetectPro*, is deployed in the commercial bank networks from BBVA-Bancomer and Banc Sabadell, in Latin America and Spain, respectively. For instance, BBVA-Bancomer's network contains the traffic of more than 12 thousand employees, 1,704 bank locations and 4,286 ATMs.

(ii) **Multimedia Traffic**: The VoIP traffic managers (described in Chapter 6), *RTPTracker* and *Skypeness*, have been tested as pilot project in Telefonica International Wholesale Services (TIWS)'s network.

7.3 Future Work

The results presented in this thesis open new research lines for future work in high-performance traffic monitoring with OTS systems. In what follows, we suggest some future research topics in this field:

- **Towards 100 Gb/s and beyond**: Although traffic monitoring of 10 Gb/s links at line-rate is considered very challenging nowadays and has been thoroughly analyzed in this thesis, scalability issues are worthy to be studied. Note that both links' speed and users' demand are rapidly evolving.

- **More advanced features of modern NICs**: The packet sniffing capabilities of contemporary NICs, such as Receive Side Scaling (RSS), have been exhaustively studied in this thesis. However, there are other features that may be still exploited. For instance, we could export to upper layers the hash computed by the NIC to map packets to RSS queues. Such operation should require only a simple modification to the NIC driver. If such hardware-computed hash is available for the

flow handling module, this will avoid to compute the hash in software, consequently reducing the CPU burden. Furthermore, the hardware filters (Flow Director) may be used to split traffic in different queues. However, the scalability and performance issues of such filters have not been analyzed.

- **Analysis of other packet timestamping issues**: This thesis has analyzed the impact of novel packet I/O engines on packet timestamping accuracy in terms of mean and variance of the error. However, other aspects of the problem have not been studied. For example, if there is autocorrelation in the observed timestamping error. Furthermore, modern NIC present some hardware timestamping capabilities. It is possible to timestamp Precision Time Protocol (PTP) packets using the NIC. However, we would like to answer the following questions: *is it possible to timestamp at line-rate? Which is the limit?*

- **High-performance active monitoring using novel OTS systems**: This thesis has thoroughly evaluated the performance of passive monitoring tasks, such as traffic classification, for high-performance and OTS systems. However, commodity hardware opens great possibilities to active monitoring that should be studied. For instance, hardware-based timestamping along with packet sniffing capabilities may be used to develop bandwidth measurement tools for high-speed links.

- **Using other OTS devices**: Non Uniform Memory Access (NUMA) systems with multi-core architectures, as well as modern multi-queue NICs have been analyzed in this thesis. However, the utilization of other commodity hardware, such as Graphic Processing Unit (GPU), has not been evaluated. Current approaches force to pass through main memory, and waste processing time, to transfer data between the NIC and the GPU creating a bottleneck. There are preliminary results that may avoid such limitation, which can be only used with Infiniband technology yet [NVI12].

- **Performance evaluation analysis of other classification meth-**

ods and monitoring tasks on OTS systems: Although we have thoroughly analyzed traffic classification (focusing on statistical methods and multimedia traffic) for high-performance and OTS systems, there are many other monitoring tasks and classification methods whose performance is worthy to be studied.

- **Evaluation of the impact of sampling and packet loss of other classification tools and classes of traffic**: This thesis assessed the impact of packet sampling on statistical Skype classification. Nevertheless, a similar study may be performed over other classification techniques and classes of traffic. For example, the analysis the impact of packet sampling on DPI-based classification techniques as well as the different effect according to the class of traffic may be relevant.

Conclusiones

Este capítulo está dedicado a resumir los principales resultados de esta tesis, a mostrar las aplicaciones industriales donde actualmente se usan tales resultados, y a dar una visión general de las direcciones previstas para el trabajo futuro para continuar con las contribuciones presentadas en este documento.

Contribuciones Principales

Esta tesis trata el problema de la clasificación de tráfico de Internet a altas prestaciones en sistemas de propósito general. Tales soluciones están basadas en hardware de uso extendido y software de código abierto. En este trabajo se estudia la viabilidad de la monitorización de red (en particular, de la clasificación de tráfico) a tasa de línea (10 Gb/s e incluso superiores) usando sistemas de propósito general. Con este fin, se analiza cada una de las partes que componen motor de clasificación de tráfico y, en general, un sistema de monitorización de red, a saber: captura de paquetes, marcado de tiempo, formación de flujos y clasificación estadística. Además, se dedica un capítulo al análisis del tráfico multimedia debido a su relevancia y popularidad en las redes actuales. Las principales conclusiones de estas contribuciones se han presentado al final de sus respectivos capítulos, sin embargo, se resumen a continuación:

(i) **Los nuevos motores de captura de paquetes propuestos en la literatura son capaces de capturar tráfico a tasas de multi-Gb/s usando sistemas de propósito general**. El capítulo 3 mostró que la utilización de hardware de uso extendido en tareas de altas

prestaciones, antes reservadas al hardware especializado, permite la captura de paquetes a tasa de línea en soluciones de propósito general. De este modo, los resultados de rendimiento presentados en este capítulo, además de la inherente flexibilidad y bajo coste de los sistemas basados en hardware de uso extendido, hace esta solución una tecnología prometedora en la actualidad. También se destaca que el análisis y desarrollo de software basado en hardware multi-núcleo es aún un problema abierto. Problemas como la agregación de flujos relacionados, el marcado de tiempo preciso y el desorden de paquetes seguro recibirán más atención por parte de la comunidad científica en el futuro. En particular, el marcado de tiempo se ha analizado en el capítulo 4 mientras que la formación de flujos se ha estudiado en el capítulo 5. Otras contribuciones de este capítulo se resumen a continuación:

- Se han identificado las limitaciones de las pilas de red y los drivers por defecto. Tales limitaciones para la captura de paquetes en los actuales sistemas operativos, desaprovechan el rendimiento potencial que podría ser obtenido usando los modernos sistemas de propósito general.
- Se han detallado las soluciones propuestas para superar tales limitaciones. En general, las claves para alcanzar alto rendimiento son la gestión eficiente de memoria, interacción con el hardware a bajo nivel y optimización de la programación. Se ha probado que el adecuado ajuste de la configuración es primordial dado el fuerte impacto que tiene sobre el rendimiento global. Nótese que este esfuerzo de revisar las limitaciones y cuellos de botella y sus respectivas soluciones pueden ser útiles para otras áreas de investigación y no solamente para tareas de monitorización o procesado de paquetes (Section 3.3).
- Se han revisado y comparado cualitativamente los diferentes motores de captura propuestos en la literatura. Se han identificado las soluciones que cada motor implementa además de sus pros y sus contras (Section 3.4).

- Se han evaluado en la misma plataforma de propósito general los diferentes motores de captura de paquetes para así analizar el rendimiento que alcanzan y su escalabilidad. En particular, se ha encontrado que cada motor puede ser más adecuado para un escenario distinto, dependiendo del rendimiento requerido y de la disponibilidad de núcleos de proceso en el sistema (Section 3.6).

Finalmente, las contribuciones de este capítulo han dado lugar a las siguientes publicaciones (presentadas en orden cronológico):

- J. Fullaondo, Pedro M. Santiago del Río, Javier Ramos, J.L. García-Dorado, and Javier Aracil, "AP-CAP framework: Monitorizando a 10 Gb/s en hardware de propósito general", in *Actas de las X Jornadas de Ingeniería Telemática (JITEL),* Santander (Spain), September 2011.
- José L. García-Dorado, Felipe Mata, Javier Ramos, Pedro M. Santiago del Río, Victor Moreno, and Javier Aracil, "Chapter 2: High-Performance Network Monitoring Systems Using Commodity Hardware", accepted for its publication in *Data Traffic Monitoring and Analysis; from measurement, classification and anomaly detection to quality of experience,* Springer, Series in Computer Communications and Networks, editors: C. Callegari, M. Matijasevic, E. Biersack, 2012.

(ii) **El marcado de tiempo preciso es viable a alta velocidad usando sistemas de propósito general**. El capítulo 4 demostró que las técnicas usadas en los nuevos motores de captura de paquetes para mejorar el rendimiento, como el procesado en lotes, introducen imprecisión en el marcado de tiempo. Se proponen tres técnicas, basadas en dos aproximaciones diferentes, para mitigar o superar tal imprecisión. Por un lado, se demuestra que distribuir (uniformemente o ponderadamente) el tiempo entre lotes entre los distintos paquetes que forman un lote, incrementa la precisión del marcado temporal. Por otro lado, un rediseño del driver de red (para constantemente consultar los buffers de

la tarjeta de red para ver si hay paquetes disponibles y luego marcarla y copiarlos a otro buffer, uno a uno) mitiga la degradación en el marcado de tiempos. Otras contribuciones de este capítulo se resumen a continuación:

- La imprecisión en el marcado temporal, que introducen los nuevos motores de captura, está sobre las decenas de microsegundos.
- La imprecisión en el marcado temporal cuando se usan nuestras propuestas se reduce significativamente. En particular, los métodos basados en la distribución del tiempo mejoran la precisión, reduciendo la desviación estándar del error por debajo de de 200 nanosegundos cuando el enlace está cargado, mientras que el métodos basados en consultar los buffer de la tarjeta de red alcanza un error de 1 microsegundo con una traza real.
- Se propone combinar ambas soluciones dependiendo de la carga del enlace, es decir, cuando el enlace está cerca de saturarse se distribuye el tiempo en grupos de paquetes y, en caso contrario, se marcan los paquetes uno a uno.

Finalmente, las contribuciones de este capítulo han dado lugar a la siguiente publicación (presentadas en orden cronológico):

- V. Moreno, Pedro M. Santiago del Río, Javier Ramos, Jaime J. Garnica, José L. García-Dorado, "Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines", *IEEE Communications Letters*, **16** (11) (2012), pp. 1888–1891.

(iii) **La monitorización y la clasificación estadística de tráfico a tasa de línea son viables usando soluciones basadas sólo en software y hardware de uso extendido**. El capítulo 5 mostró que la clasificación en tiempo real de tráfico basada en huellas estadísticas es posible gracias: al uso de un driver de red mejorado que mueve eficientemente grupos de paquetes desde la tarjeta de red a la memoria principal; al uso de técnicas ligeras de clasificación estadística que explotan el tamaño de

los primeros paquetes de cada flujo observado; y a un cuidadoso ajuste de parámetros críticos del entorno hardware y la propia aplicación software. De hecho, mientras que la tasa de clasificación en trazas real es sobre 3 veces mayor que [SGV+10] y 4 veces mayor que [VPI11], sin embargo, nuestro sistema es capaz de sostener una tasa de flujos clasificados 93veces mayor que [LKJ+10] y 560 veces mayor que [VPI11]. Más generalmente, se ha probado que la arquitectura de sistema propuesta es flexible y escalable para desarrollar sistemas de monitorización de tráfico capaces de procesar fuentes de datos de red con múltiples granularidades, a saber, a nivel de paquete de flujo y agregado, desde distintos hilos (con distintos propósitos) simultáneamente. Otras contribuciones de este capítulo se resumen a continuación:

- Cuando usamos múltiples colas y núcleos es preferible evitar la concurrencia en el acceso a estructuras de datos compartidas, como la tabla has o el anillo de trabajos. Para arbitrar tales accesos concurrente a la memoria compartida es necesario llevar a cabo operaciones de sincronización y bloqueo, que causan una gran pérdida de rendimiento (Section 5.2.5).
- Estructuras complejas como árboles RB para gestionar las colisiones no compensan, sino que es preferible una estructura que ocupe menos memoria y funciones hash cuidadosamente elegidas. En general, se puede concluir que estructuras software propuestas en el estado del arte (a saber, árboles RB equilibrados o la función hash Bon-Jenkins), no son suficientes para manejar tráfico a 10 Gb/s in el escenario de caso peor de 14,8 Mpps en un único núcleo de proceso (Section 5.2.7).
- La detallada comparación de varios métodos de aprendizaje automático señala a los árboles C4.5 como la mejor opción debido: a su conocido poder discriminatorio y al hecho de que pueden ser implementados muy eficientemente con estructuras if-else, aguantando el exigente escenario de hasta 2,8 millones de clasificaciones por segundo. Otros métodos de aprendizaje automático no son

aconsejables debido: a su menor precisión en la clasificación, como por ejemplo Naïve-Bayes o por su insuficiente rendimiento, por ejemplo SVM (Section 5.2.8).

- Se ha liberado el código del motor de clasificación propuesto y del motor de inyección de tráfico usado en los experimentos [Hrg12].

Finalmente, las contribuciones de este capítulo han dado lugar a las siguientes publicaciones (presentadas en orden cronológico):

- Pedro M. Santiago del Río, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil, "Wire-Speed Statistical Classification of Network Traffic on Commodity Hardware", in *Proceedings of ACM International Internet Measurement Conference (IMC),* Boston, MA (USA), November 2012.
- Pedro M. Santiago del Río, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil, "Early traffic classification beyond 10 Gbps using commodity hardware", under review in *Computer Communications, Elsevier.*

(iv) **La captura completa de los datos y la monitorización del tráfico de VoIP (tanto sobre SIP como el tráfico Skype) a tasas de multi-Gb/s es manejable usando sólo sistemas de propósito general**. El capítulo 6 demostró que es posible cumplir con los exigentes retos que conlleva la monitorización de tráfico multimedia en las actuales redes de alta velocidad (10 Gb/s o incluso superior). Se proponen dos novedosos sistemas modulares, para los casos de VoIP sobre SIP (Sec. 6.3) y de tráfico Skype (Sec. 6.4) respectivamente, los cuales dan muy alto rendimiento, siendo capaces de procesar tráfico al vuelo a altas tasas y con una significativa reducción del coste usando sistemas de propósito general. Otras contribuciones de este capítulo se resumen a continuación:

- No es aconsejable usar múltiples colas de recepción cuando se requiere correlar tanto los flujos SIP como los RTP porque un flujo

SIP y su flujo RTP correspondiente pueden potencialmente terminar en colas y núcleos distintos (Section 6.3.2).

- Se necesita dos tablas hash independientes (una para indexar el tráfico SIP y la otra para indexar RTP) son necesarias para gestionar el tráfico VoIP sobre SIP porque los paquetes RTP no tienen el identificador de llamado ni otra información relativa al SIP, mientras que los paquetes SIP no tienen las mismas direcciones IP ni puertos que los paquetes RTP (Section 6.3.2).
- Si se detecta en la red segmentación TCP o fragmentación IP, un módulo de reensamblado TCP/IP debe ser implementado para ser capaz de identificar el tráfico (Section 6.3.2).
- El proceso de conversión de audio (de formato raw a formato audible) se debe llevar a cabo bajo demanda cuando la conversación es solicitada por el usuario porque las tasas de conversión no son suficientes para soportar la tasa del resto del sistema en una red de 10 Gb/s, mayor de 400 llamadas cada segundo (Section 6.3.4).
- Los resultados experimentales muestra que la identificación de tráfico Skype es viable en las redes actuales de alta velocidad, típicamente en el rango de 10 a 40 Gb/s, usando hardware de propósito general. En particular, la clasificación de tráfico Skype funciona a más de 3,5 Gb/s usando un único núcleo. Este proceso escala adecuadamente, alcanzando un rendimiento de 45 Gb/s usando cuatro procesadores de 8 núcleos (Section 6.4).
- El muestreo de paquetes impacta sobre algunas características del tráfico como los tiempos entre llegadas, mientras que la distribución del tamaño de paquete permanece casi inalterada. Por tanto, la precisión de la clasificación se degrada significativamente cuando se aplica muestreo de paquetes (Section 6.5).
- Tal degradación en la clasificación de tráfico puede ser mitigada escalando los tiempos entre llegadas que se miden, dependiendo de la tasa de muestreo. De este modo, Skypeness sería capaz de clasificar a más de 300 Gb/s con una precisión destacable, dada

una tasa de muestreo de 1/8 y su capacidad de proceso de 45 Gb/s (Section 6.5).

Finalmente, las contribuciones de este capítulo han dado lugar a las siguientes publicaciones (presentadas en orden cronológico):

- Pedro M. Santiago del Río, Javier Ramos, J.L. García-Dorado, Javier Aracil, Antonio Cuadra-Sánchez, and Mar Cutanda-Rodríguez, "On the processing time for detection of Skype traffic," in *Proceedings of 7th International Wireless Communications and Mobile Computing Conference (IWCMC),* Istanbul (Turkey), July 2011.
- Pedro M. Santiago del Río, Diego Corral, José L. García-Dorado, and Javier Aracil, "On the Impact of Packet Sampling on Skype Traffic Classification," accepted for its publication in *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM),* Ghent (Belgium), May 2013.
- José L. García-Dorado, Pedro M. Santiago del Río, Javier Ramos, David Muelas, Víctor Moreno, Jorge E. López de Vergara, Javier Aracil, "Low-cost and High-performance: the case of VoIP monitoring using commodity hardware", under review in *Journal of Network and Systems Management, Springer.*

Aplicaciones Industriales

Los resultados y aplicaciones de esta tesis están siendo actualmente explotados por Naudit HPCN [CN13]. Naudit es una empresa de base tecnológica creada por una *spin-off* de dos universidades: la UAM[4] y la UPNA, y forma parte de su Campus de Excelencia Internacional[5]. En su accionariado participan, además de ambas universidades, el Consejo Superior de Investigaciones Científicas (CSIC), a través del parque científico de Madrid.

[4] `http://www.uam.es/ss/Satellite/es/1242657608103/listadoCategorizado/Spin-offs_de_la_UAM.htm`

[5] `http://campusexcelencia.uam-csic.es/`

Naudit junto con la FUAM[6] han llevado a cabo varios proyectos de innovación y transferencia tecnológica. Entre los clientes de Naudit se encuentran organismos públicos como el Ministerio de Industria del Gobierno de España, operadoras de telecomunicaciones como Movistar, grupos bancarios multinacionales como el BBVA, compañías del sector industrial como Airbus o importantes grupos energéticos.

Concretamente, los siguientes resultados de la tesis son aplicados directamente en la industria:

(i) **Análisis de tráfico y detección de anomalías**: La sonda flexible de monitorización pasiva (descrita en la sección 5.3), *DetectPro*, está desplegada en las redes bancarias comerciales de BBVA-Bancomer y Banc Sabadell, en América Latina y España, respectivamente. Por ejemplo, la red de BBVA-Bancomer contiene el tráfico de más de 12 mil empleados, 1.704 sucursales bancarias y 4.286 cajeros automáticos.

(ii) **Tráfico multimedia**: Los gestores de tráfico de VoIP (descritos en el captulo 6), *RTPTracker* and *Skypeness*, han sido probados como proyecto piloto en la red de TIWS (Telefonica International Wholesale Services).

Trabajo Futuro

Los resultados presentados en esta tesis abren nuevas líneas de investigación para trabajo futuro en la monitorización de tráfico de altas prestaciones con sistemas de propósito general. A continuación, se sugieren algunos temas de investigación futura en este área:

- **Hacia los 100 Gb/s y más allá**: Aunque la monitorización de tráfico de enlaces de 10 Gb/s a tasa de línea es considerada un reto hoy día y ha sido profundamente analizada en esta tesis, los problemas de escalabilidad merecen ser estudiados. Nótese que tanto la velocidad de los enlaces como la demanda de los usuarios están evolucionando rápidamente.

[6] `www.fuam.es`

- **Más características avanzadas de las tarjetas de red modernas**: Las capacidades de captura de paquetes de las tarjetas de red actuales, tales como las colas RSS, han sido exhaustivamente estudiadas en esta tesis. Sin embargo, existen otras características que puede ser aún explotadas. Por ejemplo, se podría exportar a capas superiores el hash calculado por la tarjeta de red para repartir paquetes en las distintas colas. Tal operación debería requerir sólo una pequeña modificación en el driver de la tarjeta. Si este hash calculado en hardware está disponible para el módulo de formación de flujos, esto evitaría calcular el hash en software, y así, reduciría la carga de CPU. Además, los filtros hardware (llamados Flow Director) pueden ser usados para dividir el tráfico en las diferentes colas. Sin embargo, los temas de escalabilidad y rendimiento de tales filtros no han sido analizados.

- **Análisis de otros aspectos del marcado de tiempos**: Esta tesis ha analizado el impacto de los nuevos motores de captura sobre la precisión del marcado temporal en términos de media y varianza del error. Sin embargo, otros aspectos del problema no han sido estudiados. Por ejemplo, si hay autocorrelación en el error de marcado obeservado. Además, las nuevas tarjetas de red presentan algunas capacidades para el marcado temporal en hardware. Es posible marcar los paquetes PTP usando la tarjeta de red. Sin embargo, nos gustaría ser capaces de responder las siguientes preguntas: *¿Es posible marcar los paquetes a tasa de línea? ¿cuál es el límite?*

- **Monitorización activa de altas prestaciones usando modernos sistemas de propósito general**: Esta tesis ha evaluado profundamente el rendimiento de tareas de monitorización pasiva, como la clasificación de tráfico, para sistemas de propósito general a altas prestaciones. Sin embargo, el hardware de uso extendido abre grandes posibilidades para la monitorización activa que deberían ser estudiadas. Por ejemplo, el marcado de tiempos basado en hardware junto con las capacidades de captura de paquetes pueden ser usados para desarrollar herramientas de medida de ancho de banda para enlaces de alta

velocidad.

- **Usar otros dispositivos de propósito general**: Los sistemas NUMA con arquitecturas multi-núcleo, además de las tarjetas de red multi-cola, han sido analizados en esta tesis. Sin embargo, la utilización de otro hardware de uso extendido, como las GPUs, no ha sido evaluada. Las actuales soluciones fuerza pasar por la memoria principal, y gastar tiempo de proceso, para transferir datos entre la tarjeta de red y la GPU, creando un cuello de botella. Hay estudios preliminares que puede evitar tal limitación, los cuales pueden ser solo usados con tecnología Infiniband todavía [NVI12].

- **Análisis de evaluación de las prestaciones de otros métodos y tareas de monitorización en sistemas de propósito general**: Aunque se ha analizado profundamente la clasificación de tráfico (centrado en los métodos estadístico y el tráfico multimedia) para sistemas de propósito general a altas prestaciones, hay muchas otras tareas de monitorización y métodos de clasificación de tráfico cuyo rendimiento merece la pena ser estudiado.

- **Evaluación del impacto del muestreo y la pérdida de paquetes de otras herramientas de clasificación y clases de tráfico**: Esta tesis ha evaluado el impacto del muestreo de paquetes de la clasificación estadística de tráfico Skype. No obstante, un estudio similar puede ser llevado a cabo sobre otras técnicas de clasificación y otras clases de tráfico. Por ejemplo, el análisis del impacto del muestreo de paquetes sobre la clasificación basada en firmas (Deep Packet Inspection (DPI)) además del efecto dependiendo de la clase de tráfico puede ser relevante.

References

[ACE09] K. Argyraky, B. Chun, and N. Egi, *RouteBRICKS*, 2009, `http://routebricks.epfl.ch/`. 42

[ACG+09] D. Adami, C. Callegari, S. Giordano, M. Pagano, and T. Pepe, *A real-time algorithm for skype traffic detection and classification*, Proceedings of the 9th International Conference on Smart Spaces and Next Generation Wired/Wireless Networking and Second Conference on Smart Spaces (St. Petersburg, Russia), NEW2AN '09 and ruSMART '09, September 2009, pp. 168–179. 129, 156

[ACG+12] _____, *Skype-hunter: A real-time system for the detection and classification of skype traffic*, International Journal of Communication Systems **25** (2012), no. 3, 386–403. 127

[AKA08] A. Agrawal, K.R.P. Kumar, and G. Athithan, *SIP/RTP session analysis and tracking for VoIP logging*, Proceedings of the 16th IEEE International Conference on Networks (New Delhi, India), ICON'08, December 2008, pp. 1–5. 25

[AL13] Alcatel-Lucent, *FP3: breakthrough 400G network processor*, 2013, `http://www.alcatel-lucent.com/fp3/`. 15

[AZH08] D. Angevine and A.N. Zincir-Heywood, *A preliminary investigation of Skype traffic classification using a minimalist feature set*, Proceedings of the 3rd International Conference

on Availability, Reliability and Security (Barcelona, Spain), ARES'08, March 2008, pp. 1075–1079. 127, 151

[AZH09] R. Alshammari and A.N. Zincir-Heywood, *Machine learning based encrypted traffic classification: Identifying SSH and Skype*, Proceedings of the 2nd IEEE Symposium on Computational Intelligence for Security and Defense Applications (Ottawa, Canada), CISDA'09, July 2009, pp. 1–8. 127, 151

[BDKC10] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, *Comparing and improving current packet capturing solutions based on commodity hardware*, Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (Melbourne, Australia), IMC'10, November 2010, pp. 206–217. 2, 130

[BDPGP12] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, *On multi-gigabit packet capturing with multi-core commodity hardware*, Proceedings of the 13th International Conference on Passive and Active Network Measurement (Vienna, Austria), PAM'12, March 2012, pp. 64–73. 40, 65

[BDPP13] N. Bonelli, A. Di Pietro, and G. Procissi, *PFQ project*, 2013, `https://github.com/pfq/PFQ`. 48

[BHA09] P. A. Branch, A. Heyde, and G. J. Armitage, *Rapid identification of Skype traffic flows*, Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video (Williamsburg, VA, USA), NOSSDAV'09, June 2009, pp. 91–96. 129

[BMM+07] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli, *Revealing Skype traffic: when randomness plays with you*, ACM SIGCOMM Computer Communication Review **37** (2007), no. 4, 37–48. 119, 126, 149, 156

[BMM+08] D. Bonfiglio, M. Mellia, M. Meo, N. Ritacca, and D. Rossi, *Tracking down Skype traffic*, Proceedings of the 27th IEEE Conference on Computer Communications (Phoenix, AZ, USA), INFOCOM'08, April 2008, pp. 261–265. 25

[BMMR09] D. Bonfiglio, M. Mellia, M. Meo, and D. Rossi, *Detailed analysis of Skype traffic*, IEEE Transactions on Multimedia **11** (2009), no. 1, 117 –127. 119, 127

[BMPR10] R. Birke, M. Mellia, M. Petracca, and D. Rossi, *Experiences of VoIP traffic monitoring in a commercial ISP*, International Journal of Network Management **20** (2010), no. 5, 339–359. 3, 118, 142, 146

[Bon94] J. Bonwick, *The slab allocator: An object-caching kernel memory allocator*, Proceedings of the USENIX Summer 1994 Technical Conference (Boston, MA, USA), USTC'94, June 1994, pp. 6–6. 20

[BRV09] T. Broomhead, J. Ridoux, and D. Veitch, *Counter availability and characteristics for feed-forward based synchronization*, Proceedings of the 3rd IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication (Brescia, Italy), ISPCS'09, October 2009, pp. 1–6. 65

[BTS06] L. Bernaille, R. Teixeira, and K. Salamatian, *Early application identification*, Proceedings of the 2nd ACM Conference on Future Networking Experiments and Technologies (Lisbon, Portugal), CoNEXT'06, December 2006, pp. 6:1–6:12. 24, 25, 28, 78, 99

[CAI11] CAIDA, *CoralReef*, 2011, `http://www.caida.org/tools/measurement/coralreef/`. 25, 92

[CAI12] CAIDA, *Traffic analysis research*, 2002-2012, `http://www.caida.org/data/passive/trace_stats/`. 54

[CCR11] N. Cascarano, L. Ciminiera, and F. Risso, *Optimizing deep packet inspection for high-speed traffic analysis*, Journal of Network and Systems Management **19** (2011), no. 1, 7–31. 126

[CDGS07] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, *Traffic classification through simple statistical fingerprinting*, ACM SIGCOMM Computer Communication Review **37** (2007), no. 1, 5–16. 24, 25, 28, 78, 81, 99, 100

[CEBRCASP11] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Sol-Pareta, *Analysis of the impact of sampling on Netflow traffic classification*, Computer Networks **55** (2011), no. 5, 1083–1099. 120

[Cis09] Cisco, *Monitoring VoIP with Cisco network analysis module*, October 2009, `http://www.cisco.com/en/US/prod/collateral/modules/ps2706/white_paper_c11-520524_ps2706_Products_White_Paper.html`. 15, 130

[CN13] Naudit High Performance Computing and Networking, 2013, `http://www.naudit.es`. 178, 190

[CPB93] K.C. Claffy, G.C. Polyzos, and H.-W. Braun, *Application of sampling methodologies to network traffic characterization*, ACM SIGCOMM Computer Communication Review **23** (1993), no. 4, 194–203. 160, 161

[CR04] B. Caswell and M. Roesch, *Snort: The open source network intrusion detection system*, 2004. 23

[CSK11] Y. Choi, J. Silvester, and H.-C. Kim, *Analyzing and modeling workload characteristics in a multiservice IP network*, IEEE Internet Computing **15** (2011), no. 2, 35–42. 118, 141

[CSMBMG07] M. Cardenete-Suriol, J. Mangues-Bafalluy, A. Maso, and M. Gorricho, *Characterization and comparison of Skype behavior in wired and wireless network scenarios*, Proceedings of the 50th IEEE Global Communications Conference (Washington, D.C, USA), GLOBECOM'07, November 2007, pp. 2167–2172. 128

[DAR12] M. Dobrescu, K. Argyraki, and S. Ratnasamy, *Toward predictable performance in software packet-processing platforms*, Proceedings of 9th USENIX Conference on Networked Systems Design and Implementation (San Jose, CA, USA), NSDI'12, April 2012, pp. 11–11. 12

[DB09] L.H. Do and P. Branch, *Real time VoIP traffic classification*, Tech. Report 090914A, Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, 2009, `http://caia.swin.edu.au/reports/090914A/CAIA-TR-090914A.pdf`. 129

[DC12] L. Deri and A. Cardigliano, *libzero for PF_RING DNA*, 2012, `http://www.ntop.org/products/pf_ring/libzero-for-dna/`. 43

[DCM10] L. De Cicco and S. Mascolo, *A mathematical model of the Skype VoIP congestion control algorithm*, IEEE Transactions on Automatic Control **55** (2010), no. 3, 790 –795. 119, 127

[DCMP07] L. De Cicco, S. Mascolo, and V. Palmisano, *An experimental investigation of the congestion control used by Skype VoIP*, Proceedings of the 5th international conference on Wired/Wireless Internet Communications (Coimbra, Portugal), WWIC'07, May 2007, pp. 153–164. 127

[DCMP08] ———, *Skype video responsiveness to bandwidth variations*, Proceedings of the 18th ACM International Workshop on

Network and Operating Systems Support for Digital Audio and Video (Braunschweig, Germany), NOSSDAV'08, May 2008, pp. 81–86. 127

[DDDS11] M. Danelutto, L. Deri, and D. De Sensi, *Network monitoring on multicores with algorithmic skeletons*, Proceedings of the 30th International Conference on Parallel Computing (Ghent, Belgium), PARCO'11, August 2011. 20

[DEA+09] M. Dobrescu, N. Egi, K. Argyraki, B-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, *Routebricks: exploiting parallelism to scale software routers*, Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (Big Sky, MT, USA), SOSP'09, October 2009, pp. 15–28. 40

[Der08] L. Deri, *IP traffic monitoring at 10 Gbit and above*, 2008, `http://www.terena.org/activities/ngn-ws/ws2/deri-10g.pdf`. 20

[Dig13] Digium, *Asterisk*, 2013, `http://www.asterisk.org`. 141

[DPC12] A. Dainotti, A. Pescapè, and K.C. Claffy, *Issues and future directions in traffic classification*, IEEE Network **26** (2012), no. 1, 35 –40. 2, 24, 28

[dT06] Telecommunication Networks Group Politecnico di Torino, *Skype traces: Traces from real internet traffic*, 2006, `http://tstat.tlc.polito.it/tracce/Polito/2006/11_01_29_May_SKYPE_UDP_E2E.dump.anonim.gz`. 154, 161

[EGS09] Alice Este, Francesco Gringoli, and Luca Salgarelli, *Support vector machines for TCP traffic classification*, Elsevier Computer Networks **53** (2009), no. 14, 2476–2490. 98

[End12] Endace, *DAG cards: network traffic recording*, 2012, `http://www.endace.com`. 14

[FD10] F. Fusco and L. Deri, *High speed network traffic analysis with commodity multi-core systems*, Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (Melbourne, Australia), IMC'10, November 2010, pp. 218–224. 40, 103

[FK11] S. Frankel and S. Krishnan, *RFC 6071: IP security (IPSec) and internet key exchange (IKE) document roadmap*, 2011. 21

[FMM+11] A. Finamore, M. Mellia, M. Meo, M.M. Munafò, and D. Rossi, *Experiences of Internet traffic monitoring with Tstat*, IEEE Network **25** (2011), no. 3, 8 –14. 19, 20, 103, 126, 148

[FMMR10] A. Finamore, M. Mellia, M. Meo, and D. Rossi, *KISS: Stochastic packet inspection classifier for UDP traffic*, IEEE/ACM Transactions on Networking **18** (2010), no. 5, 1505–1515. 25, 27

[Fou12] Linux Foundation, *NAPI: The New API*, 2012, `http://www.linuxfoundation.org/collaborate/workgroups/networking/napi`. 16

[FS93] J. Fenlason and R. Stallman, *Gnu gprof: The GNU profiler*, 1993, `http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html`. 89

[GDMnR+12] J.L. García-Dorado, E. Magaña, P. Reviriego, M. Izal, D. Morató, J.A. Maestro, J. Aracil, and J.E. López de Vergara, *Network monitoring for energy efficiency in large-scale networks: the case of the spanish academic network*, The Journal of Supercomputing **62** (2012), 1284–1304. 2

[GES12] F. Gringoli, A. Este, and L. Salgarelli, *MTCLASS: Traffic classification on high-speed links with commodity hardware*, Proceedings of the IEEE International Conference on Communications (Ottawa, Canada), ICC'12, June 2012, pp. 1177 –1182. 19, 103

[GJ13] R. Gayraud and O. Jacques, *SIPp*, 2013, `http://sipp.sourceforge.net/`. 141

[God12] S. Godard, *Sysstat*, 2012, `http://sebastien.godard.pagesperso-orange.fr/`. 86

[Goo02] B. Goode, *Voice over Internet protocol (VoIP)*, Proceedings of the IEEE **90** (2002), no. 9, 1495–1517. 121, 124

[GS78] L.J. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, Proceedings of the 19th Annual Symposium on Foundations of Computer Science (Ann Arbor, MI, USA), FOCS'78, October 1978, pp. 8–21. 94

[HB08] T. Hossfeld and A. Binzenhofer, *Analysis of Skype VoIP traffic in UMTS: End-to-end QoS and QoE measurements*, Computer Networks **52** (2008), no. 3, 650 – 666. 119, 128

[HJ98] M. Handley and V. Jacobson, *RFC 2327: SDP: Session description protocol*, 1998. 25, 122

[HJPM10] S. Han, K. Jang, K. Park, and S. Moon, *PacketShader: a GPU-accelerated software router*, ACM SIGCOMM Computer Communication Review **40** (2010), 195–206. 2, 10, 14, 21, 23, 32, 33, 36, 40, 42, 51, 65, 78, 79, 103, 110, 157

[HJPM12] ———, *Packetshader I/O engine*, 2012, `http://shader.kaist.edu/packetshader/io_engine/index.html`. 45

[Hrg12] Universidad Autónoma de Madrid HPCN research group, *HPTRAC: High Performance TRAffic Classifier*, 2012,

http://www.eps.uam.es/~psantiago/hptrac.html. 79, 84, 175, 188

[HSZ09] C. Henke, C. Schmoll, and T. Zseby, *Empirical evaluation of hash functions for PacketID generation in sampled multi-point measurements*, Proceedings of the 10th International Conference on Passive and Active Network Measurement (Seoul, South Korea), PAM'09, April 2009, pp. 197–206. 93

[IEE08] IEEE Instrumentation and Measurement Society, *IEEE Standard for a precision clock synchronization protocol for networked measurement and control systems*, 2008. 12

[Int12] Intel, *82599 10 Gbe controller datasheet*, 2012, http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html. 10

[Int13] ———, *Ixp4xx product line of network processors*, 2013, http://www.intel.com/p/en_US/embedded/hwsw/hardware/ixp-4xx. 15

[IT10] C.M. Inacio and B. Trammell, *YAF: yet another flowmeter*, Proceedings of the 24th USENIX International conference on Large installation system administration (San Jose, CA, USA), LISA'10, November 2010, pp. 1–16. 20

[JLM+12] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K.S. Park, *Kargus: a highly-scalable software-based intrusion detection system*, Proceedings of the 19th ACM Conference on Computer and Communications Security (Raleigh, NC, USA), CCS'12, October 2012, pp. 317–328. 19, 36, 103

[KBB+04] T. Karagiannis, A. Broido, N. Brownlee, K.C. Claffy, and M. Faloutsos, *Is P2P dying or just hiding? [P2P traffic measurement]*, Proceedings of the 47th IEEE Global Communications Conference (Dallas, TX, USA), GLOBECOM'04, November 2004, pp. 1532–1538. 25

[KCF+08] H. Kim, K.C. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K.Y. Lee, *Internet traffic classification demystified: myths, caveats, and the best practices*, Proceedings of the 4th ACM Conference on Future Networking Experiments and Technologies (Madrid, Spain), CoNEXT'08, December 2008, pp. 11:1–11:12. 24, 28, 99, 100

[Ker12] A.D. Keromytis, *A comprehensive survey of voice over IP security research*, IEEE Communications Surveys & Tutorials **14** (2012), no. 2, 514 –537. 2, 130

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek, *The Click modular router*, ACM Transactions on Computer Systems **18** (2000), no. 3, 263–297. 21, 40

[KP09] S. Karapantazis and F.N. Pavlidou, *VoIP: A comprehensive survey on a promising technology*, Computer Networks **53** (2009), no. 12, 2050–2090. 3, 118, 124

[KPF05] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, *BLINC: multilevel traffic classification in the dark*, ACM SIGCOMM Computer Communication Review **35** (2005), no. 4, 229–240. 25, 27

[Kra09] M. Krasnyansky, *UIO-IXGBE*, 2009, `https://opensource.qualcomm.com/wiki/UIO-IXGBE`. 40

[LCSR11] M. Laner, S. Caban, P. Svoboda, and M. Rupp, *Time synchronization performance of desktop computers*, Proceedings of the 5th IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication

(Munich, Germany), ISPCS'11, September 2011, pp. 75–80. 66

[Lf09] L7-filter, *Application layer packet classifier for linux*, 2009, `http://l7-filter.sourceforge.net`. 25

[LK09] A.B. Lim and R. Kinsella, *Data plane packet processing on embedded intel architecture platforms*, 2009, `http://www.intel.com/content/www/us/en/intelligent-systems/wireless-infrastructure/ia-data-plane-packet-processing-paper.html`. 20

[LKJ+10] Y. Lim, H. Kim, J. Jeong, C. Kim, T.T. Kwon, and Y. Choi, *Internet traffic classification demystified: on the sources of the discriminative power*, Proceedings of the 6th ACM Conference on Future Networking Experiments and Technologies (Philadelphia, PA, USA), CoNEXT'10, November 2010, pp. 9:1–9:12. 24, 25, 28, 78, 99, 100, 101, 102, 116, 175, 187

[LSI12] LSI: Storage, Networking, Accelerated, *Network processors*, 2012, `http://www.lsi.com/products/networkingcomponents/Pages/networkprocessors.aspx`. 15

[LXSL08] Y. Liu, D. Xu, L. Sun, and D. Liu, *Accurate traffic classification with multi-threaded processors*, Proceedings of the IEEE International Symposium on Knowledge Acquisition and Modeling Workshop (Wuhan, China), KAM'08, December 2008, pp. 478–481. 26, 78

[LZB11] G. Liao, X. Znu, and L. Bnuyan, *A new server I/O architecture for high speed networks*, Proceedings of the 17th Symposium on High-Performance Computer Architecture (San Antonio, TX, USA), HPCA'11, February 2011, pp. 255–265. 32, 33

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *Openflow: enabling innovation in campus networks*, ACM SIGCOMM Computer Communication Review **38** (2008), no. 2, 69–74. 21

[MGDA12] F. Mata, J.L. García-Dorado, and J. Aracil, *Detection of traffic changes in large-scale backbone networks: The case of the spanish academic network*, Computer Networks **56** (2012), no. 2, 686 – 702. 109

[Mic13] Microsoft, *Receive Side Scaling*, 2013, `http://msdn.microsoft.com/en-us/library/windows/hardware/ff567236(v=vs.85).aspx`. 10

[MKK+01] D. Moore, K. Keys, R. Koga, E. Lagache, and K.C. Claffy, *The CoralReef software suite as a tool for system and network administrators*, Proceedings of the 15th USENIX International conference on Large installation system administration (San Diego, CA, USA), LISA'01, December 2001, pp. 133–144. 25, 92

[MNB07] A. Mitra, W. Najjar, and L. Bhuyan, *Compiling PCRE to FPGA for accelerating SNORT IDS*, Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (Orlando, FL, USA), ANCS'07, December 2007, pp. 127–136. 26

[MSdRR+12] V. Moreno, P.M. Santiago del Río, J. Ramos, J.J. Garnica, and J.L. García-Dorado, *Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines*, IEEE Communications Letters **16** (2012), no. 11, 1888–1891. 38, 40, 60

[NA08] T.T.T. Nguyen and G. Armitage, *A survey of techniques for Internet traffic classification using machine learning*, IEEE

Communications Surveys & Tutorials **10** (2008), no. 4, 56–76. 3, 24, 27, 28, 81, 97, 99, 120, 126

[NVI12] NVIDIA Corporation, *NVIDIA GPUDirect Technology*, 2012, `http://developer.download.nvidia.com/devzone//devcenter/cuda/docs/GPUDirect_Technology_Overview.pdf`. 115, 180, 193

[oB09] Telecommunication Networks Group University of Brescia, *UNIBS: data sharing*, 2009, `http://www.ing.unibs.it/ntw/tools/traces`. 98, 99

[PCA10] PCAP, *tcpdump & libpcap*, 2010, `http://www.tcpdump.org/`. 30, 46

[Pro13] Open MPI Project, *Portable hardware locality (hwloc)*, 2013, `http://www.open-mpi.org/projects/hwloc/`. 152

[PV07] P. Perälä and M. Varela, *Some experiences with VoIP over converging networks*, Proceedings of the Measurement of Speech, Audio and Video Quality in Networks workshop (Prague, Czech Republic), MESAQIN'07, June 2007. 128

[PVA+12] A. Papadogiannakis, G. Vasiliadis, D. Antoniades, M. Polychronakis, and E.P. Markatos, *Improving the performance of passive network monitoring applications with memory locality enhancements*, Computer Communications **35** (2012), no. 1, 129–140. 32

[QXH+07] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, *Towards high-performance flow-level packet processing on multi-core network processors*, Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (Orlando, FL, USA), ANCS'07, December 2007, pp. 17–26. 20

[RCC12] L. Rizzo, M. Carbone, and G. Catalli, *Transparent acceleration of software packet forwarding using netmap*, Proceedings of the 31th IEEE Conference on Computer Communications (Orlando, FL, USA), INFOCOM'12, March 2012, pp. 2471–2479. 21

[RDC12] L. Rizzo, L. Deri, and A. Cardigliano, *10 Gbit/s line rate packet processing using commodity hardware: survey and new proposals*, 2012, `http://luca.ntop.org/10g.pdf`. 40

[Riz12a] L. Rizzo, *netmap: a novel framework for fast packet I/O*, Proceedings of the 2012 USENIX conference on Annual Technical Conference (Boston, MA, USA), ATC'12, jun 2012, pp. 101–112. 17, 32, 38, 40, 54, 55

[Riz12b] ———, *netmap project*, 2012, `http://info.iet.unipi.it/~luigi/netmap/`. 47

[Riz12c] ———, *Revisiting network I/O APIs: the netmap framework*, Communications of the ACM **55** (2012), no. 3, 45–51. 103

[RM06] D. Rossi and M. Mellia, *Real-time TCP/IP analysis with common hardware*, Proceedings of the IEEE International Conference on Communications (Istanbul, Turkey), ICC'06, June 2006, pp. 729–735. 20

[RSC+02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *RFC 3261: SIP: Session initiation protocol*, 2002. 25, 122

[RVV+08] D. Rossi, S. Valenti, P. Veglia, D. Bonfiglio, M. Mellia, and M. Meo, *Pictures from the Skype*, ACM SIGMETRICS Performance Evaluation Review **36** (2008), no. 2, 83–86. 81, 100

[SCFJ] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, *RFC 3550: RTP: A transport protocol for real-time applications*. 137

[Sch12] H. Scholz, *IETF Internet-Draft: RTP stream information export using IPFIX*, 2012. 139

[SdRHA+10] P.M. Santiago del Río, J.A. Hernández, J. Aracil, J.E. López de Vergara, J. Domzal, R. Wójcik, P. Cholda, K. Wajda, J.P. Fernández Palacios, Á. González de Dios, and R. Duque, *A reliability analysis of Double-Ring topologies with Dual Attachment using p-cycles for optical metro networks*, Computer Networks **54** (2010), no. 8, 1328 – 1341. 2

[SdRRG+12] P.M. Santiago del Río, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil, *Wire-speed statistical classification of network traffic on commodity hardware*, Proceedings of the 12th ACM conference on Internet measurement conference (Boston, MA, USA), IMC '12, November 2012, pp. 65–72. 70, 103

[SGI08] G. Stampfel, W. N. Gansterer, and M. Ilger, *Implications of the EU data retention directive 2006/24/EC*, Proceedings of Sicherheit (Saarbrucken, Germany), April 2008, pp. 45–58. 3, 119, 130, 139

[SGV+10] G. Szabó, I. Gódor, A. Veres, S. Malomsoky, and S. Molnár, *Traffic classification over Gbit speed with commodity hardware*, IEEE Journal of Communications Software and Systems **5** (2010), no. 3. 23, 26, 78, 102, 175, 187

[SHR+09] P. Svoboda, E. Hyytiä, F. Ricciato, M. Rupp, and M. Karner, *Detection and tracking of Skype by exploiting cross layer information in a live 3G network*, Proceedings of the First International Workshop on Traffic Monitoring

and Analysis (Aachen, Germany), TMA '09, May 2009, pp. 93–100. 128

[SoX13] SoX, *Sound eXchange*, 2013, `http://sox.sourceforge.net/`. 141

[SWF07] F. Schneider, J. Wallerich, and A. Feldmann, *Packet capture in 10-Gigabit Ethernet environments using contemporary commodity hardware*, Proceedings of the 8th International Conference on Passive and Active Measurement Conference (Louvain-la-neuve, Belgium), PAM'07, April 2007, pp. 207–217. 140

[Sys12] Cisco Systems, *Introduction to Cisco IOS Netflow - a technical overview*, 2012, `http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/prod_white_paper0900aecd80406232.pdf`. 154

[SZTG12] W. Su, L. Zhang, D. Tang, and X. Gao, *Using direct cache access combined with integrated NIC architecture to accelerate network processing*, Proceedings of the 14th IEEE Conference on High Performance Computing and Communications and the 9th IEEE Conference on Embedded Software and Systems (Liverpool, UK), HPCC-ICESS'12, June 2012, pp. 509–515. 40

[TB11] B. Trammell and E. Boschi, *An introduction to IP flow information export (IPFIX)*, IEEE Communications Magazine **49** (2011), no. 4, 89 –95. 108

[Tcp12] Tcpreplay, *Pcap editing & replay tools for *NIX*, 2012, `http://tcpreplay.synfin.net/`. 84, 156, 158

[TVRP12] D. Tammaro, S. Valenti, D. Rossi, and A. Pescapè, *Exploiting packet-sampling measurements for traffic characterization and classification*, International Journal of Network Management **22** (2012), no. 6, 451–476. 120

[Uni13] Stanford University, *NetFPGA*, 2013, `http://netfpga.org`. 14

[VGBR09] S. Varadarajan, S. Gopalan, V.S. Basavaraja, and K.K. Rao, *System and method for Skype traffic detection. U.S. patent 20090116394*, May 2009, `http://www.freepatentsonline.com/y2009/0116394.html`. 127

[VM11] N. Varis and J. Manner, *In the network: Sandy Bridge versus Nehalem*, ACM SIGMETRICS Performance Evaluation Review **39** (2011), no. 2, 53–55. 42

[VPI11] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis, *MIDeA: a multi-parallel intrusion detection architecture*, Proceedings of the 18th ACM Conference on Computer and Communications Security (Chicago, IL, USA), CCS'11, October 2011, pp. 297–308. 14, 19, 22, 23, 36, 78, 102, 103, 115, 175, 187

[WAcA09] C. Walsworth, E. Aben, k.c. claffy, and D. Andersen, *The CAIDA anonymized 2009 Internet traces*, 2009, `http://www.caida.org/data/passive/passive_2009_dataset.xml`. 53, 72, 78, 89, 111

[WDC11] W. Wu, P. DeMar, and M. Crawford, *Why can some advanced Ethernet NICs cause packet reordering?*, IEEE Communications Letters **15** (2011), no. 2, 253–255. 33, 35, 60, 65, 83, 132

[WP12] S. Woo and K. Park, *Scalable TCP session monitoring with Symmetric Receive-Side Scaling*, Tech. report, Department of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Seoul, South Korea, 2012, `http://www.ndsl.kaist.edu/~shinae/papers/TR-symRSS.pdf`. 11, 37

[WXD11] D. Wang, Y. Xue, and Y. Dong, *Memory-efficient hypercube flow table for packet processing on multi-cores*, Proceedings of the 54th IEEE Global Communications Conference (Houston, TX, USA), GLOBECOM'11, December 2011, pp. 1–6. 20

[YZ10] J. Yu and X. Zhou, *Ultra-high-capacity DWDM transmission system for 100G and beyond*, IEEE Communications Magazine **48** (2010), no. 3, S56–S64. 2, 118

[ZFP12] L. Zabala, A. Ferro, and A. Pineda, *Modelling packet capturing in a traffic monitoring system based on Linux*, Proceedings of 2012 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (Genoa, Italy), SPECTS'12, July 2012, pp. 1–6. 16

[ZLQ+12] H. Zhang, G. Lu, M.T. Qassrawi, Y. Zhang, and X. Yu, *Feature selection for optimizing traffic classification*, Computer Communications **35** (2012), no. 12, 1457–1471. 99

[Zob70] A.L. Zobrist, *A new hashing method with application for game playing*, International Computer-Chess Association Journal **13** (1970), no. 2, 69–73. 26

[ZR10] T. Zourzouvillys and E. Rescorla, *An introduction to standards-based VoIP: SIP, RTP, and friends*, IEEE Internet Computing **14** (2010), no. 2, 69 –73. 121

List of Publications

Publications Directly Related to this Thesis

1. Pedro M. Santiago del Río, Javier Ramos, J.L. García-Dorado, Javier Aracil, Antonio Cuadra-Sánchez, and Mar Cutanda-Rodríguez, "On the processing time for detection of Skype traffic," in *Proceedings of 7th International Wireless Communications and Mobile Computing Conference (IWCMC),* Istanbul (Turkey), July 2011.
   Chapter 6 in this thesis.

2. J. Fullaondo, Pedro M. Santiago del Río, Javier Ramos, J.L. García-Dorado, and Javier Aracil, 'AP-CAP framework: Monitorizando a 10 Gb/s en hardware de propósito general," in *Actas de las X Jornadas de Ingeniería Telemática (JITEL),* Santander (Spain), September 2011.
   Chapter 3 in this thesis.

3. Pedro M. Santiago del Río, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil, "Wire-Speed Statistical Classification of Network Traffic on Commodity Hardware," in *Proceedings of ACM International Internet Measurement Conference (IMC),* Boston, MA (USA), November 2012.
   Chapter 5 in this thesis.

4. V. Moreno, Pedro M. Santiago del Río, Javier Ramos, Jaime J. Garnica, José L. García-Dorado, "Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines.,"', *IEEE Communications Letters*, **16** (11) (2012), pp. 1888–1891.

Chapter 4 in this thesis.

5. Pedro M. Santiago del Río, Diego Corral, José L. García-Dorado, and Javier Aracil, "On the Impact of Packet Sampling on Skype Traffic Classification.," accepted for its publication in *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM),* Ghent (Belgium), May 2013.
   Chapter 6 in this thesis.

6. José L. García-Dorado, Felipe Mata, Javier Ramos, Pedro M. Santiago del Río, Victor Moreno, and Javier Aracil, "Chapter 2: High-Performance Network Monitoring Systems Using Commodity Hardware," accepted for its publication in *Data Traffic Monitoring and Analysis; from measurement, classification and anomaly detection to quality of experience,* Springer, Series in Computer Communications and Networks, editors: C. Callegari, M. Matijasevic, E. Biersack, 2012.
   Chapter 3 in this thesis.

7. Pedro M. Santiago del Río, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil, "Early traffic classification beyond 10 Gbps using commodity hardware", under review in *Computer Communications, Elsevier.*
   Chapter 5 in this thesis.

8. José L. García-Dorado, Pedro M. Santiago del Río, Javier Ramos, David Muelas, Víctor Moreno, Jorge E. López de Vergara, Javier Aracil, "Low-cost and High-performance: the case of VoIP monitoring using commodity hardware", under review in *Journal of Network and Systems Management, Springer.*
   Chapter 6 in this thesis.

Publications in Topics Related to this Thesis

1. Javier Ramos, Pedro M. Santiago del Río, Javier Aracil, and Jorge E. López de Vergara, "On the effect of concurrent applications in band-

width measurement speedometers,"', *Computer Networks*, **55** (6) (2011), pp. 1435–1453.

2. Pedro M. Santiago del Río, José A. Hernández, Javier Aracil, Jorge E. López de Vergara, Jerzy Domzal, Robert Wojcik, Piotr Cholda, Krzysztof Wajda, Juan P. Fernández-Palacios, Óscar González de Dios, and Raúl Duque, "A reliability analysis of Double-Ring topologies with Dual Attachment using p-cycles for optical metro networks,"', *Computer Networks*, **54** (8) (2010), pp. 1328–1341.

3. Pedro M. Santiago del Río, Javier Ramos, Alfredo Salvador, Jorge E. López de Vergara, Javier Aracil, Antonio Cuadra-Sánchez, and Mar Cutanda-Rodríguez, "Application of Internet Traffic Characterization to All-Optical Networks," in *Proceedings of 12th International Conference on Transparent Optical Networks (ICTON),* Munich (Germany), June 2010.

4. Javier Aracil, Javier Ramos, Pedro M. Santiago del Río, Jorge E. López de Vergara, Luis de Pedro, Sergio López-Buedo, Iván González, and Francisco J. Gómez-Arribas, "Método para estimar los parámetros de un elemento de control de tipo Tocken-Bucket (METHOD FOR ESTIMATING THE PARAMETERS OF A CONTROL ELEMENT SUCH AS A TOKEN BUCKET)," Patent ES 2372213 A1, Spain, 09/04/2010.

5. Pedro M. Santiago del Río, José A. Hernández, Víctor López, Javier Aracil, and Bas Huiszoon, "On the feasibility of transmission scheduling in a code-based transparent passive optical network architecture," in *Proceedings of 14th European Conference on Networks and Optical Communications (NOC),* Valladolid (Spain), June 2009.

Index