UNIVERSITÁ DI PISA
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

HIGH-PERFORMANCE NETWORK PROGRAMMING FOR MULTICORE ARCHITECTURES

DOCTORAL THESIS

Author
Nicola Bonelli

Tutor (s)
Prof. Stefano Giordano
Dr. Gregorio Procissi

Reviewer (s)
Prof. Giuseppe Bianchi
Prof. Sandrine Vaton

The Coordinator of the PhD Program
Prof. Marco Luise

Pisa, October 2017

Cycle XXX

This thesis is dedicated to my family and friends

"The greatest obstacle to discovery is not ignorance.
It is the illusion of knowledge."
Daniel J. Boorstin

Acknowledgements

I would like to thank the CNIT consortium for the chance I have been given to fuel my passion and pursue my ambitious sense of research within the BEBA project, and all the partners involved for their proverbial availability.

I am also grateful to all those people who have supported me in these years and have endured my critical thinking in order to achieve the best possible results.

I would like to express my gratitude to the staff of the Netgroup Pisa, my tutors Gregorio and Stefano, my reviewer Giuseppe and Sandrine, my colleagues (from UniRoma and PoliMi), and especially to my parents, my brother, Erika and Cheyenne, for supporting me during the writing of this thesis.

Ringraziamenti

Ringrazio il consorzio CNIT e tutti i partner coinvolti nel progetto BEBA per la loro proverbiale disponibilità e per avermi concesso la possibilità di accrescere la passione per la ricerca e perseguire così il mio ambizioso obiettivo.

Sono inoltre grato a tutte quelle persone che, nel corso di questi anni, mi hanno supportato ed hanno sopportato il mio senso critico per consentirmi di ottenere i migliori risultati possibili.

Infine esprimo la mia gratitudine al personale del Gruppo Reti di Pisa, ai miei tutor Gregorio e Stefano, ad i revisori Giuseppe e Sandrine, ed ai miei colleghi di UniRoma e PoliMi. In particolar modo ringrazio i miei genitori, la mamma Giovanna ed il babbo Vittorugo, mio fratello Sandro, Erika e Cheyenne per avermi sostenuto durante la stesura di questa tesi.

Summary

OVER the last few years the Internet has become a pervasive network that interconnects billions of users and heterogeneous devices. The speed at which services mutate and the increasing number of devices (e.g., the advent of Internet Thing) call for innovative tools of network management and monitoring.

This rapid evolution poses new challenges for the scientific community and the industry, which are facing similar problems, though with different objectives.

The scientific community has moved to seek accelerated software solutions for the Internet traffic management on low-cost platforms. The industry, to meet new market demands, is called to develop more and more efficient products, with flexibility and reconfigurability as primary goals.

The choice to replace old and expensive equipment with personal computers (PC), has allowed achieving better performance and functionality, reducing both the development times and the production costs. Consequently, the cooperation between the scientific community and industry has enabled the creation of two new technologies dedicated to the development of the network and its services: SDN (software defined network) and NFV (network virtualization function).

In this context, the thesis delves into network programming for multi-core architectures, with the goal to identify and formalize the techniques required to compete with the growing speed of the network and the often underused parallelism of modern commodity hardware.

Most of the general purpose languages adopted for the development of network applications are still inadequate, due to the complexity of parallel programming. Therefore developers fail to create applications with the goal of performance, security, and flexibility at the same time.

On this premise, we set ourselves the goal of making a functional framework for the Linux operating system, equipped with a specific programming language, allowing a rapid development of high-performance network applications in a safe and straightforward manner.

In chapter 1 we introduce the reader to the efficient network programming, and from that point of view, we eviscerate in the gory details the technicalities of concurrency and

parallelism in modern architectures.

The chapters that follow revolve around the key aspects of such a topic.

In chapter 2 we describe the internals of the PFQ framework and in particular its kernel-bypass architecture, the programming user interfaces and the performance obtainable with its use in network applications.

Later, in chapter 3 we tackle the foundation of traffic distribution and highlight the benefit deriving from the adoption of PFQ as an accelerated engine to enable parallel execution of legacy applications. In particular, to exploit the multi-processor architectures, we analyze an extension for the standard libpcap interface with fanout algorithms based on the "divide and conquer" principle.

Chapter 4 introduces the reader to the DSL (Domain Specific Language) area and presents a prototyped functional language designed for both stateless and stateful processing of network packets. In short, we start with the theoretical background of functional languages and category theory (monads); we give a formal description of the grammar, and then we evaluate two implementations, for both the kernel- and user-space, in term of flexibility and performance.

Finally, in chapter 5 we show some use cases and applications related to high-performance network programming.

We present the BEBA flavor OpenFlow Soft Switch, in which PFQ is used as an accelerated engine for packet switching. The acceleration obtained – quantified with a factor between 90 and 100 – is the combined effect of the improvements achieved on a single core with the parallelism of multiple core architectures.

Afterwards, we describe a scenario where PFQ is deployed as a tool to perform measurements on a 10Gbit network, with the aim to prove the correctness of an algorithm for enforcing fair-bandwidth among many TCP-like senders. In this case, the use of the framework is necessary to carry out non-invasive measurements not affecting the normal operation of the Linux kernel stack.

In the end, we present a system designed for anomaly detection where the framework is used as a building block of multiple distributed probes intended to collect traffic from a backbone network and detect different kinds of anomalies. In particular, we present the library that implements a set of composable probabilistic counters which are used to spot heavy hitters, taking advantage of their fundamental aggregation property.

Sommario

NEL corso degli ultimi anni Internet è diventata una rete pervasiva che collega miliardi di utenti e dispositivi tra loro eterogenei. La velocità con cui i servizi vengono aggiornati combinata con l'aumento del numero dei dispositivi di rete (Internet of Thing) richiede pertanto nuovi strumenti dedicati alla gestione ed il monitoraggio.

Questa rapida evoluzione pone nuove sfide sia alla comunità scientifica che all'industria, che si trovano ad affrontare problematiche simili, seppur con obiettivi diversi.

La comunità scientifica è mossa a cercare soluzioni software accelerate per la gestione del traffico Internet su piattaforme di basso costo. L'industria, invece, che deve far fronte alle esigenze di mercato, sviluppa prodotti sempre più performanti, con la necessità di mantenere flessibilità e riconfigurabilità tra i principali obiettivi.

La scelta di sostituire i vecchi e costosi apparati con i moderni personal computer (PC), ha consentito di raggiungere funzionalità e prestazioni migliori, abbattendo sia i tempi di sviluppo che i costi di produzione. Conseguentemente, la collaborazione tra comunità scientifica ed industria ha permesso anche la nascita di nuove tecnologie specifiche per lo sviluppo della rete ed i servizi correlati: SDN (software defined network) e NFV (network function virtualization).

In questa tesi viene approfondito il tema della programmazione di rete per architetture multi-core, con l'obiettivo di identificare e formalizzare le tecniche necessarie per competere con le crescenti velocità delle reti e sfruttare il parallelismo del moderno hardware di consumo.

La maggior parte dei linguaggi general purpose adottati per lo sviluppo di applicazioni di rete, come ad esempio il linguaggio C, risulta infatti inadeguata a causa della complessità della programmazione parallela. Per questa ragione gli sviluppatori non sempre riescono a creare applicazioni performanti, sicure e flessibili al tempo stesso.

Sulla base di questa premessa, la tesi si pone come obiettivo quello di realizzare un framework funzionale per il sistema operativo Linux, equipaggiato con un linguaggio di programmazione specifico, che consenta un rapido sviluppo di applicazioni di rete ad alte prestazioni, in modo semplice e sicuro.

Nel capitolo 1 viene introdotto il lettore alla pratica della programmazione di rete efficiente, e sono presentate nel dettaglio alcune tecniche avanzate di concorrenza e parallelismo necessarie per raggiungere prestazioni elevate su moderne architetture di calcolatori.

I capitoli che seguono ruotano intorno agli aspetti chiave di questo argomento.

Nel capitolo 2 vengono descritti i dettagli interni del framework PFQ, ed in particolare l'architettura del kernel-bypass, le interfacce utente di programmazione e le prestazioni ottenibili mediante il suo utilizzo in applicazioni di rete.

Successivamente, nel capitolo 3 vengono affrontati i principi della distribuzione del traffico ed evidenziati i maggiori vantaggi derivanti dall'adozione di PFQ come motore per l'esecuzione parallela di applicazioni esistenti. In particolare, viene descritta l'estensione dell'interfaccia pcap con algoritmi di fanout fondati sul principio "divide et impera" per sfruttare al meglio le architetture multi processore.

Il capitolo 4 introduce il lettore nel mondo del DSL (Domain Specific Language) presentando il prototipo di un linguaggio funzionale per l'elaborazione dei pacchetti di rete. Partendo dal background teorico dei linguaggi funzionali e dalla teoria delle categorie (monadi), viene fornita una descrizione formale della grammatica, con semplici esempi e casi d'uso. Successivamente vengono presentate e valutate due implementazioni del linguaggio, una specifica per lo spazio kernel ed una per lo spazio utente.

Nel capitolo 5 vengono quindi mostrati alcuni casi d'uso ed applicazioni correlate alla programmazione di rete ad elevate prestazioni.

In particolare, viene prestato uno switch software compatibile OpenFlow (BEBA Soft Switch), in cui PFQ viene impiegato come motore per l'accelerazione della commutazione dei pacchetti. Le prestazioni del prototipo presentano un fattore di accelerazione valutato tra 90x e 100x, e risultano essere l'effetto combinato dei miglioramenti ottenuti su un singolo core con il parallelismo di più processori.

Successivamente, viene descritto uno scenario in cui PFQ implementa uno strumento di misura su una rete a 10 Gbit, al fine di dimostrare per via sperimentale la correttezza di un algoritmo di fair-bandwidth tra sorgenti TCP-like multiple. In questo caso il framework è impiegato per effettuare misure non invasive, ovvero che non vadano ad influenzare il normale funzionamento dello stack di rete del kernel di Linux.

Infine, viene illustrato un sistema progettato per l'individuazione di anomalie di rete. In particolare viene presentata una libreria che implementa un set di contatori statistici componibili che vengono utilizzati come blocco funzionale per realizzare sonde destinate alla raccolta del traffico di una rete backbone. L'obiettivo in questo caso è quello di rilevare anomalie e heavy hitter, sfruttando le proprietà di aggregabilità di tali contatori.

List of publications

International Journals

1. Bonelli, N., Giordano, S., Procissi, G. (2016). Network traffic processing with pfq. *IEEE Journal on Selected Areas in Communications*, 34(6), 1819-1833.
2. Bonelli, N., Giordano, S., Procissi, G. and Del Vigna, F. (2017). Packet fanout extension for the standard pcap library. *IEEE Transaction on Network and Service Management*. Currently under second cycle review.
3. Bonelli, N., Callegari, C., Procissi, G. (2017). A Probabilistic Counting Framework for Distributed Measurements. *IEEE Transactions on Network and Service Management*. Submitted.
4. Fernandes, E., Rothenberg, C., Bonelli, N., Kis, Z. L.,Rojas, E. and Sanvito, D. (2017). SDN prototyping gone wild: Advancing the state of art through the implementation of open networking standard. *ACM Sigcomm Computer Communication Review*. In submission.

International Conferences/Workshops with Peer Review

1. Cascone, C., Bonelli, N., Bianchi, L., Capone, A., Sanso', B. (2017,Jun). Towards Approximate Fair Bandwidth Sharing via Dynamic Priority Queuing. *IEEE International Symposium on Local and Metropolitan Area Networks*. IEEE ComSoc.
2. Bonelli, N., Giordano, S., Procissi, G. (2017, June). Enabling packet fan-out in the libpcap library for parallel traffic processing. In *Network Traffic Measurement and Analysis Conference (TMA)*, 2017 (pp. 1-9). IEEE.
3. Bonelli, N., Giordano, S., Procissi, G. (2017, July). A pipeline functional language for stateful packet processing. In Network Softwarization (NetSoft), 2017 IEEE Conference on (pp. 1-4). IEEE.
4. Bonelli, N., Procissi, G., Sanvito, D., and Bifulco, R. (2017, Nov). The acceleration of OfSoftSwitch. *IEEE Conference on Network Function Virtualization and Software Defined Networks*. IEEE.

Invited talks

1. Bonelli, N. (2016,Sep). Functional Network Programming. *Tyrrhenian International Workshop on Digital Communications*. Cnit.
2. Bonelli, N. (2017,Feb). Software Accelerations for Network Applications. *Technicolor Workshop on Network Virtualization*. IRISA (Technicolor).

Contents

Introduction

The volume of traffic crossing the Internet is continuously growing, with recent reports from Cisco [34] forecasting up to 3.3 Zettabytes of network traffic by 2021. The heterogeneity of traffic is rapidly increasing as well, due to the adoption and spreading out of new technologies, protocols, and services, bringing new requirements that can hardly be met by traditional systems and applications.

In the same vein, network applications in charge of performing any processing on real data must be able to handle vast volumes of heterogeneous traffic on communication links. This is commonly the case of network monitoring applications, Intrusion Detection and Prevention Systems, routers, firewall and so forth, that must operate online with packet arrivals and process data in streaming mode to catch—up with traffic pace and promptly trigger the necessary operations. Finally, the network scenario itself is changing, with speed of links easily hitting 10+ Gbps and increasing host density in large data centers.

Applications in charge of processing such amounts of data (for example, traffic monitoring applications) are becoming more and more complex, not only for the service they are called to provide but also for the increasing speeds of networks they have to cope with.

A common belief that upgrading the hardware of a server does suffice to improve the performance of the software is often misleading. While it can be accurate for standard applications, such as those specific for the scientific calculus or entertainment, it is almost never the case for network applications, for which a hardware upgrade usually brings little if nothing in term of performance improvements.

The reason why network applications cannot take a direct advantage of a hardware upgrade must be sought in the software and the way it interacts with the hardware. It turns out that most of the already existing network applications were designed in the last two decades, while the advent of the new technologies, including multi-processor architectures and network adapters with multi-queues, is relatively new.

As a consequence, the software can no longer be developed with the techniques of the 1990s, but it demands the adoption of more complicated semantics (e.g., multi-threading) that requires modification to the source code and the rewriting of those parts interacting with the hardware.

Under this light, a network application is made of many software layers, which include not only the user algorithms but also the network drivers. Although this definition contrasts with that of a classic computer science – where a clear separation is placed between user space and the operating system – for us, a network application consists of multiple software layers; the one that interacts with the network cards, the abstractions built on top of it (e.g., socket and the user libraries) and the source code of the application itself.

As the performance is one of the primary goals for a modern network application, this thesis focuses on how to improve such kind of software at best and take advantage of the hardware capabilities, all in a simple, safe and reusable manner.

All the software layers are the object of our research, including the network device drivers, the abstraction layer at kernel space that handles network packets, the way such packets are delivered to the network stack or userspace queues, and how applications process them. Everything is done under the fundamental constraint of not twisting the architecture of the applications or sacrificing the compatibility with the plethora of hardware already supported by the operating systems.

As such, we decided not to come up with handcrafted versions of network device drives just to improve the pure I/O performance. Instead, we have privileged the hard part, the study and the understanding of what the real impairments are in the network context, with the final aim to develop a general and competitive framework comparable with other prominent studies (e.g., PF_RING, DPDK, and Netmap).

We propose a software framework specifically tailored to capture, transmit, and distribute network traffic to multiple instances or threads of network applications, and we show how this solution can be used efficiently in real-world scenarios, running on multi-core architectures.

Such a framework is designed to implement new applications natively or can be plugged transparently into already existing ones, without significant modifications to the source codes.

Furthermore, we argue that general purpose programming languages, such as C or C++, are no longer suitable for implementing network applications in a safe and efficient manner.

Chapter 1 raises the light on this topic, presenting the relatively new development abstractions necessary to build high-performance applications. Starting from the lesson learned these years, in this chapter we show how difficult it can be to write software for multi-core architecture using general purpose languages.

Given the recent rise of functional languages, we propose two companion languages explicitly tailored for networking.

A seminal functional approach towards flexible and programmable networks was made by SwitchWare [12]. However, back in 1998, the performance of SwitchWare suffered from the limitations of the underlying OCaml language. In our work, instead, we aim at proving that the functional approach may well reach top class performance, just as close as the ones that can be attained by imperative style programming.

We come up with two different implementations, the second one as a natural evolution of the first, running either in kernel- or user-space.

The first one is equipped with a run-time interpreter and plugged as a functional engine inside a Linux kernel module, while the second is implemented as pure embedded DSL in the Haskell language, that can be compiled as a stand-alone library and executed in standard applications.

CHAPTER 1

Elements of Network Programming

Computer network programming involves writing computer programs which enable processes to communicate with each other across a computer network. A broader definition also covers those applications that interact with the network, not necessarily with the sole purpose of communication. Network Monitoring applications, Network Intrusion Detection Systems (NIDS), router, firewalls, to mention a few, are typical examples of network applications too.

Such applications are mostly designed to run on a general purposes architecture, and now that personal computers are getting faster and faster, commodity hardware has gained a significant position in both research and industry, playing a role that is all but marginal.

For a network application, most challenging aspects comprise the speed of the network, the capabilities of the hardware and the way the software handles the two. Although it can be obvious, what makes a network application unique in comparison to a standard one is that it is designed to process network packets.

There are real-world contexts where the rate of packets is exceptionally high. This is due not only to the high-speed of the networks (10/40 and 100 Gbps) but also to the average packet length of certain protocols (the shortest the length is, the higher the rate can be). A typical example is that of VoIP, where, depending on the codec in use, the average length ranges between 100 and 250 bytes. In such a scenario, the inter-arrival rate can be rather high. It does suffice to apply the trivial formula of the ethernet throughput [1] to realize that a 10Gbps link can transport several millions of VoIP packets per second (about 5 to 9 Mpps).

Another aspect to consider is that non-trivial applications can spend several CPU cycles to handle a packet. Such a cost includes the capture, the pure processing, and

[1] https://en.wikipedia.org/wiki/Ethernet_frame

the optional re-transmission of the packet itself – though not all network applications are designed to re-transmit the traffic captured.

In each of these stages, the majority of the cycles are spent for I/O operations, either for the retrieval of the packet's payload or the state associated with the flow it belongs to (TCP, UDP flows or any other broad abstraction). Such operations are hugely slow compared to the billions of instructions that a modern multi-core CPU can process per second.

The faster the speed of the network, the higher the concurrent number of flow transported – a 40G LTE link can indeed transport dozens of millions of user flows. Let's assume to have a tracking application that associates a state to each stream (which can be either a simple counter or something more complicated). This application can result in handling a huge hash table with millions of entries (flow states) indexed by a canonical tuple – e.g., a 5-tuple, as in protocol, IP addresses source and destination, port source and destination. Because the flows of packets are usually interlaced each other, this also results in a continuous cache invalidation, as the CPU has to retrieve the state associated with each flow, on a per-packet basis.

Furthermore, the CPUs have nearly reached the maximum frequencies possible; it goes without saying that network applications have a per-core limit (in term of packet per seconds processed) which is often well below the rate offered by the network. As an example, a non-trivial application that spends thousands of CPU cycles per packet and hardly handles more than 3 million packets per second even running on a 3 GHz CPU core.

The very challenge of a network application is to cope with the packet rate offered by fast networks. In turn, this shifts into the necessity to exploit the modern hardware, which nowadays includes NUMA multi-processor architectures, the multi-core CPUs, and the network cards equipped with multiple hardware queues (e.g., Intel 82599 10G NIC).

By doing so, a network application is asked to scale linearly with the number of core involved in the processing stage. To achieve this theoretical result, a software engineer has to solve a long series of problems and impediments.

This thesis seeks to bring light to this challenge, by proposing effective solutions that can also be used in other fields of computer engineering. The following sections summarize such challenges, by dealing with some in-depth and postponing the discussion of others to the chapters that follow.

1.1 Parallel processing

The primary challenge for a network engineer is that of parallel programming, a necessary but not a sufficient condition for applications to use and scale with all cores of a CPU.

Historically Unix programs are multi-process based, but modern applications tend to use the relatively new approach of multi-threading. The main difference is that processes are implemented with a clear separation of memory, whereas threads share anything with each other.

Apart from this, it is worth mentioning that there are no other significant differences. As of today, both are implemented in kernel space on top of processes (Linux kernel), and the performance achievable from the two are mostly the same – even the cost of forking a process and creating a thread is similar.

As in a company where employees work together, a multi-threaded application can take advantage of multiple contexts, possibly running each on a separate core, to process data, produce output, and exchange messages with true parallelism.

Unfortunately, many applications are still written as a single process, and the conversion to a multi-threaded design requires a significant rework of their internals.

In an ideal world, a thread processes incoming packets in isolation, without requiring any communication or synchronization with its siblings. To achieve this a certain prepartition of the traffic is required.

The reader will often encounter the concept of state-aware traffic split. We will refer to this concept as the ideal partitioning strategy that lets each thread of execution process packets, streams, and related states in complete isolation.

Differently, threads that share data structures incur in a race condition when one (or more of them) performs an update (usually a write to memory) while others are accessing it. Since race conditions lead to undefined behaviors, a solution to this problem is to adopt a mutex to serialize the access to the shared resource.

Unfortunately, as we will show in the section 1.5 mutexes, or their most advanced counterpart spinlocks, affect the performance in two essential respects. First, they prevent applications to scale linearly with the number of cores, as all threads are therein serialized, with potential and critical side effects, such as the priority inversion phenomena. Second, the interlocked operations necessary to guarantee the correctness of a mutex are operation expensive per se and should be avoided as rule of the thumb.

1.2 Atomics operations

CPUs provide a set of lower-level operations that are not affected by data race condition. These operations are called atomic because they can not be interrupted, and in this regard, they guarantee the correct execution on multi-processor architectures.

It is possible to avoid mutual exclusion using atomic operations. Indeed, when a thread performs an atomic operation, other threads see it as happening instantaneously.

Depending on the CPU, the set of atomic instructions may vary. In modern architectures (e.g., Intel), it includes reads and writes to a given memory location, increments and decrements, the primary mathematics instructions – such as add, sub, multiplication, and so forth – and the fundamental "exchange" and "compare and swap" (CAS) operations.

The main advantage of atomic operations is that they are relatively quick compared to locks. However, in the contest of network programming, even atomic operations are expensive. As a result, a simple 'atomic' counter shared among multiple threads can dramatically affect the overall performance of a system, especially in case of high contention. The example reported in section 1.5.1 shows how a counter can be efficiently implemented in order not to affect the performance of concurrent incrementing, on a per-packet basis.

Atomic operations also represent the primary building-blocks for particular data types called lock-less data structures. A lock-less structure is a collection of information shared among multiple threads of execution that allows concurrent online updates while other threads are visiting it.

Such a structure is always well defined as the atomic operations – used to access or manipulate it – guarantee that updates to elements are safe and atomic. A typical example of a lock-free data structure is that of the queue; a mechanism often used to allow efficient message passing schemes across multiple threads.

1.3 Memory layout

A second but not less important factor is that of memory layout. How to accommodate data in memory becomes fundamental in high-performance applications.

The reason should be sought in the current technology with which modern processors are built. It is well known that CPUs are equipped with few memory levels (we refer to these with L1, L2, and L3), which are distinguished by size and speed – read and write latency. In the case of the Intel Xeon processor, for example, L1 is a cache of a few KB, very fast (a read costs 3 or 4 CPU cycles), L2 is an intermediate level cache, and L3 is the largest (24 MB in the current versions) but the slowest. There is also the central memory which, compared to the various cache is huge (Gigabytes) but about 2 or 3 orders of magnitude slower.

There are also NUMA architectures that replicate the same 3-tier model in multiple packages. NUMA introduces impairments in inter-process communications when the threads are pinned on cores of different packages since the data being exchanged between the two, across the L3 caches involve more slow communication channels.

The effect of sharing data between threads impacts on the memory in many respects. Every time a core updates the value of a global variable, for instance, it triggers the update of its local cache, the actual write to the central memory (possibly deferred) and the cache invalidation of other cores. This mechanism is governed by the cache coherence algorithm, which guarantees the uniformity of shared resource data stored in multiple local caches.

The granularity with which a cache memory maps a memory region is known as a cache line, and consist of 64 bytes of contiguous, aligned memory – 128 if speculative memory prefetch is enabled. Every time a single byte is written in one of such addresses the invalidation of the corresponding line is triggered for all caches in the system.

If such a write is fundamental for the communication between the threads of execution, the effect cannot be avoided and the delay – the time necessary for such a byte to travel from a core to another – is a latency that we have to pay – though it can be amortized with techniques involving batch processing.

Conversely, if such a write is unnecessary for the communication, the pure cache invalidation is an impairment that can be avoided. This effect is known as *false sharing* and applies when threads unconsciously impact the performance of each other while modifying independent variables sharing the same cache line. As a rule of thumb, independent variables accessed by different threads should be accommodated in distinct cache lines.

1.4 Memory allocations

Another aspect of no less importance is the dynamic memory allocation.

In a network application, the performance related to the fast data path is affected by an improper use of memory allocators, as the cost of each single operation – allocation and deallocation pair – in the current implementations is about 200-500 CPU clock cycles. If this number does not appear too large, one should think that on a 3Ghz CPU there are only 1000 clock cycles available to handle a packet at a rate of 3 million per second [2].

That said, a network application should not use dynamic memory allocation at all, dot. Even if this can be relatively easy to achieve in new applications, in already existing ones, it could mean rewriting much of the software that handles packets from scratch.

However, only those applications that can consume packets online can entirely avoid using dynamic memory. Applications that require the lifetime of the packet content exceed the time of its processing and cannot get rid of it – e.g., when the payload must be stored for later use, just like in IP defragmentation.

For these applications, we can try to reduce the number of memory allocations, as we will see later in the chapter 5, or use allocators particularly optimized for multi-core architectures (i.e., Jemalloc or TCmalloc).

1.5 Two practical examples

In this section, we present two further examples that demonstrate a direct application of the elements analyzed so far. In particular, we first analyze the performance of a simple counter shared among a set of threads which is incremented by all of them. Subsequently, we present a comparison of some lock-free queue implementations that can be used in a context of a single producer and a single consumer thread (SPSC).

1.5.1 Counters

Counters are often the only tool available to evaluate the behavior of a network application from the field. They are very lightweight mechanisms for instrumenting an application without affecting performance, and as such, they stand out from any other debugging mechanism. However, due to the operating mechanisms of memory cache that govern a CPU, even a simple counter can affect the performance of a multi-threaded application.

In this section, we will compare the performance achievable by adopting different types of counters, and we will come up with an implementation that has an almost negligible impact in term of performance.

The table below reports the list of counters engaged in this test.

counter	description
atomic	simple atomic counter
mutex	simple counter protected by a mutex
spinlock	simple counter protected by a ticket spinlock
sparse	simple sparse counter
no-sharing	simple sparse counter without false sharing

The first one is the atomic counter, that is a global variable shared among some threads, accessed for individual increments by atomic operations. We compare such a

[2]the maximum packet rate on a 10G link is indeed 14.88 Mpps

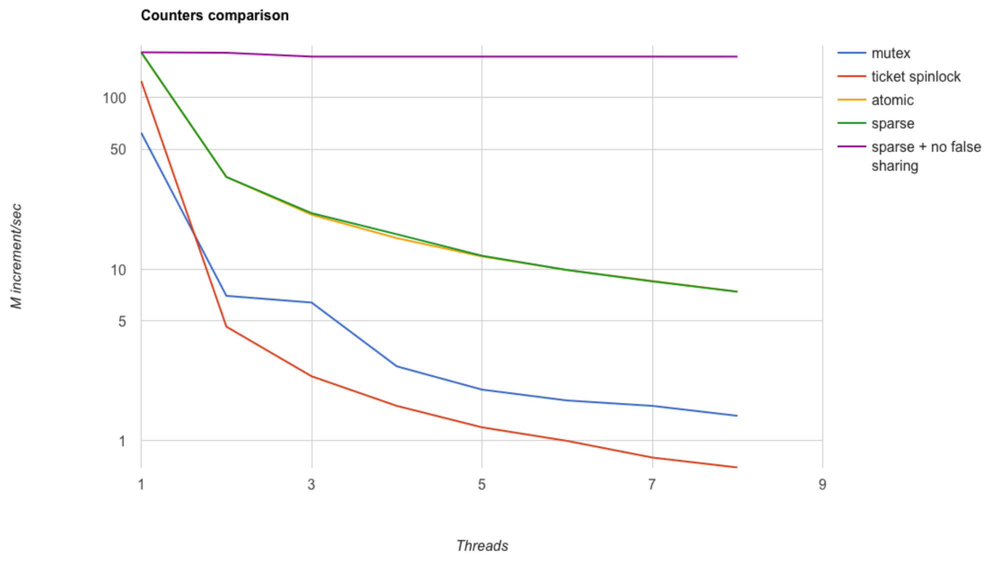


Figure 1.1: *Counters: performance comparisons*

counter with a simple counter protected by a mutex in one case, and with the lightweight spinlock in the other.

In the test, we have also included two additional counters, whose components are distributed in memory and where each thread of processing is intended to increment its component. For this type of counter, the overall value can be roughly obtained by summing up the value of all components – such operation happens asynchronously from time to time and hence is not perfect.

The difference between the two is that the first allocates the components of each thread in a continuous region of memory, while the second accommodates each component in a different cache line. The graph in figure 1.1 reports the maximum number of million of increments per second (using a log scale) each thread can handle, by varying the number of threads involved.

The graph shows that the distributed version with no false sharing is the only counter that can guarantee the independence of each thread, that can increase the component without affecting the performance of others.

Similarly, the graph in figure 1.2 represents the maximum possible rate of increments for the global counter – still in log fashion and millions per second. This chart is even more interesting in that it highlights two essential aspects, nonvisible in the first graph. First, that only the distributed counter without false-sharing can scale with the number of threads (the trend is logarithmic for the scale); second, that for all the other counters the maximum rate worsens as soon as the thread count is higher than one, and in some cases falls below the 10 million per second.

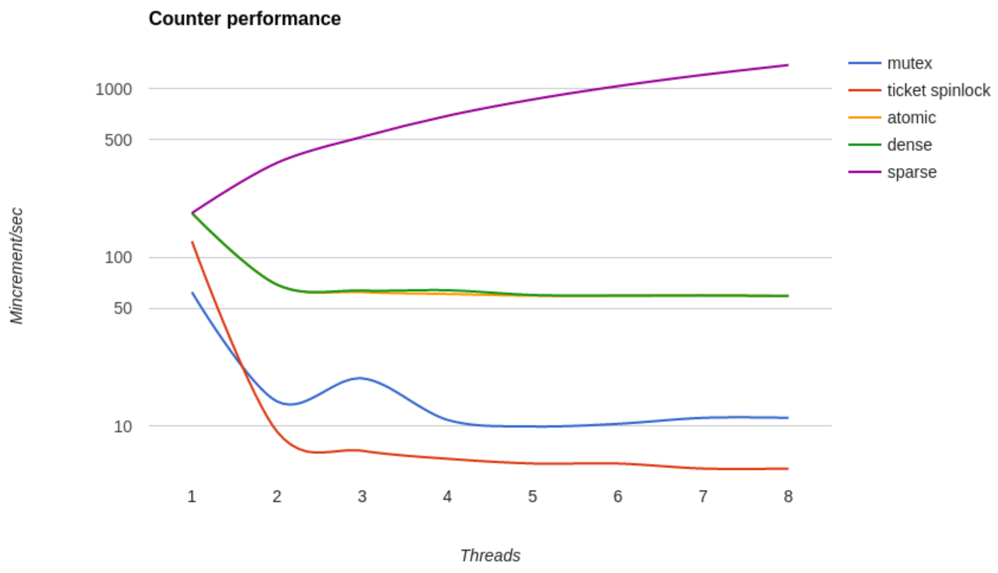


Figure 1.2: *Counters: performance comparisons*

1.5.2 Lock-less queues

Lock-less queues are data structures shared between two or more threads that allow message passing and whose implementation does not make use of any locking.

Albeit these queues are a relatively new research object, different implementations depend on the architecture and the number of threads that can use them concurrently. As expected, not all of them offer the same degree of flexibility and performance – More details are available at the site [3].

There are facilities with a single consumer and producer (SPSC), other variants with either multiple-producer or consumer (SPMC or MPSC), and queues with multiple producers and consumers (MPMC).

SPSC is the most straightforward lock-less queue, as the Intel architecture allows a trivial implementation for it. Because the compiler usually aligns integers to memory, and the hardware is not allowed to reorder writes at different memory addresses, it turns out that such a kind of queue can be implemented without atomic operations – for the sake of fussiness they require only a compiler barrier. This fact has contributed a lot to its popularity, although the implementation without atomics is not portable over different architectures – e.g., popular versions do not work on the ARM.

The queue, depicted in figure 1.3 consists of a circular ring of elements (in this context pointers to packet descriptors) and a couple of indices, one for the producer and one for the consumer, each incremented (with the modulo operation to handle wraparound) at the insertion or the removal of an element, respectively.

Each operation always requires reading both the indices since the push method is supposed to detect when the queue is full, while the pop when it is empty – possibly

[3]http://www.1024cores.net/

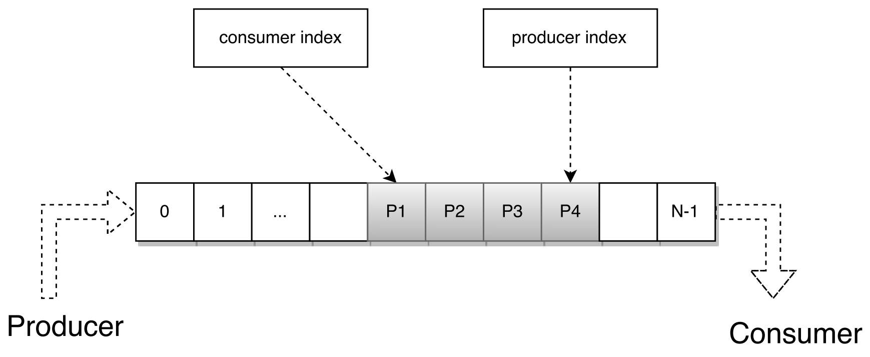


Figure 1.3: *SPSC queue scheme*

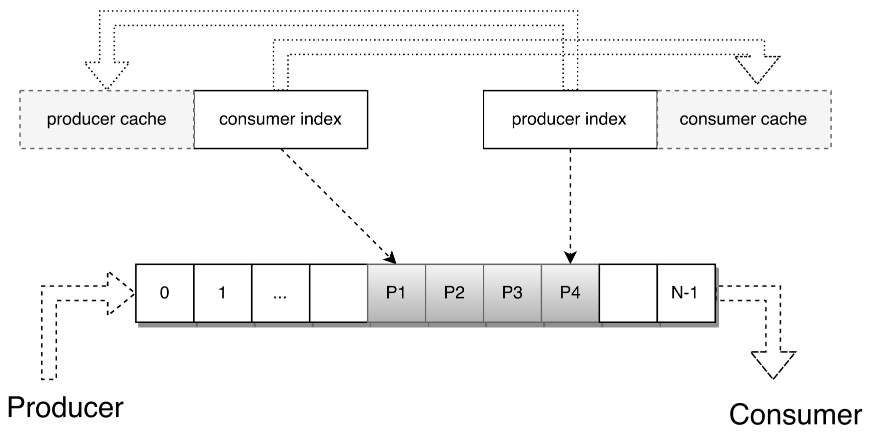


Figure 1.4: *SPSC variant, with cached indices*

to raise an error. Due to the continuous ping-pong for the CPU cache invalidations – which take place at each insertion or extraction –, the trivial implementation is fast, but not the fastest.

For this reason, we designed an improved version of this queue represented in figure 1.4. Such an implementation takes advantage of the fact that, in a cache line reserved for each, producer and consumer retain a copy of the other's index. This fact significantly improves the performance of the queue, as each thread can independently pop or push elements without involving the reading of its counterparty index – which happens from time to time when the queue appears full or empty. Performance is improved by a factor 2.2 concerning the basic queue – 310 millions of insertions and removal per second versus 138 million.

Additional improvements are brought by the adoption of a buffer which delays the publication of the indexes for both producer and consumer. From an experimental measure 1.5, it appears that a delay of 32 elements is sufficient to increase the performance and that 64 or 128 are enough to reach the maximum possible on our architecture, which is about 500 million insertions/removal per second.

Although these numbers may seem high, in the economy of a high-performance network application they can still represent a significant cost factor. In absolute terms,

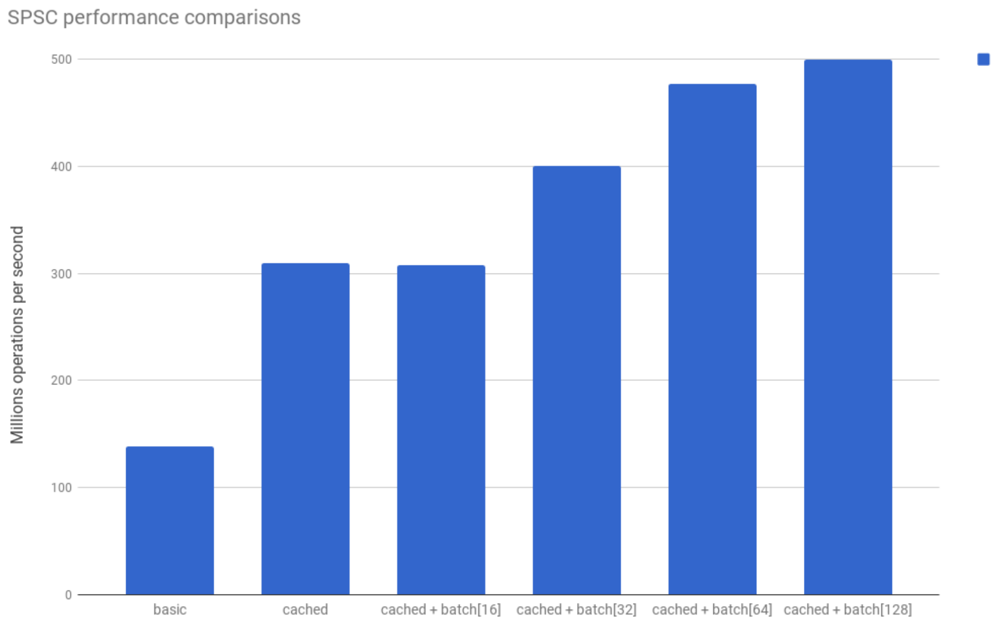


Figure 1.5: *SPSC queues: performance comparisons*

the cost of an insertion and extraction for the first SPSC queue is about 20-25 clock cycles (estimation for a 3Ghz CPU), which falls to 6 CPU clocks for the queue equipped with cache and the batch buffer.

For an application designed to perform lightweight processing at very high speed, even 20 CPU cycles can introduce a significant performance degradation.

1.6 Efficient data structures

Beside the programming best practices above introduced, a significant processing speed-up can be attained by adopting fast and memory-efficient data structures. Indeed, all monitoring applications have a specific “ultimate” purpose. This purpose can range from the need to early detect an attack, to the need to control that traffic flows or aggregates do not exceed a negotiated rate, to the need of recognising some traffic and prevent that this abuses of the available capacity, and so on. To accomplish these goals, an application could have to perform several but straightforward tasks. For instance, a network attack is often preceded by an analysis performed by the attacker to gather essential information (e.g. ARP scan, port scan, and so forth). The detection of such scanning in a monitored traffic aggregate suggests that a more in-depth analysis should be performed on the flows that generate such patterns. Similarly, to understand if a traffic flow exceeds a given packet rate, it may seem necessary first to measure the rate of all the traffic flows and then determine (and control) the ones that fail to meet the pre-established rate constraints.

Frequently, in carrying out such “intermediate” tasks, monitoring applications need to waste computational resources in processing an unnecessarily enormous amount of information. If a flow is not performing any port scanning activity, and if the application is designed to inspect only scanning flows, then the details of the non-scanning flows (such as their identifiers) are not needed. Preferably, the only information strictly needed is that flows identified as, say, A, B, and C are indeed performing scans while all the others are not: the knowledge of the identifiers A, B, C triggering a more in-depth analysis and reactive mechanisms.

In most cases, it turns out that a performance-effective solution is to split the operations of an application according to a *two–stages* approach. In this paradigm, the first stage performs a first eventually rough, but very fast, analysis of the data traffic with the goal of discriminating and isolating flows which are not of utility to the application, from the possibly few ones which instead should be further inspected. This latter category includes flows which may exhibit possible anomalous behaviour, which raise the suspect of attacks or intrusions, and in most generality flows or patterns which are specific targets for the monitoring application. The second processing stage receives as input only the flows or the per-flow information isolated by the first stage and performs the required analysis, more in-depth.

In both phases, but especially in the first one, the adoption of a system paradigm aware of the available hardware is a crucial element to attain high performance. Indeed, to accomplish operations like capturing, classification, anomaly detection, flow discrimination and isolation at the wire speed, a detailed knowledge of the hardware capabilities/bottlenecks, as well as the fine-grained analysis of the available time budget for each micro-operation involved are required. In particular, even if CPU speed is nowadays typically adequate, memory is still a very critical resource, and the proper use of small but remarkably fast cache memories may dramatically boost the overall application performance.

The attempt to come up with performance effective solutions must then pursue the investigation of *stateless* and *memory saving* approaches in that they tightly reflect into faster operations. Therefore, a large room for research is open to any proposals that can efficiently provide a smart – and possibly statistical – processing of data while requiring a reasonable level of memory utilisation. In this scenario, a very promising approach towards packet processing and inspection is based on Bloom Filters (BFs) [18] and their variations. BFs are compact and fast data structures for approximated set–membership query and their popularity is rapidly increasing because of their very limited memory requirements (roughly speaking, they implement the principle of “trading certainty for time/space” [110]). A BF represents a set of n elements by using a bitmap of m elements. Each element of the set is mapped to k elements of the bitmap whose position is given by the result of k hash functions. To check whether an element belongs to the set one just needs to evaluate the k hash functions and verify if the corresponding bits of the bitmap are all set. Naturally, the filter allows for false positives, in that the hash functions of different elements may collide. Nevertheless, a proper choice of both the length of the bitmap and the number of hash functions minimises the probability of false positive. However, the use of BFs for sets whose elements may change over time is not recommendable in that deletion of elements from the bitmap is not allowed. Counting Bloom Filters (CBFs) (or, similarly, sketches) [45] are simple extensions that replace the bits of the bitmap with counters (bin). This way, insertions and deletions of elements are as straightforward as incrementing/decrementing the value of the counter. The use of CBFs for statistical data processing turns out to be extremely flexible although, in its original version, the fixed size of bins causes in many realistic cases an unnecessary waste of memory. A significant improvement can be obtained by allowing the dynamic size of bins, compressions, and multi-layering (as in [46, 47]). For example, let us think about a set of rules to be checked by the front-end classifier: a CBF can easily be used to represent the set. To verify whether a packet obeys one of the rules of the set, a simple lookup operation, which consists of evaluating k hash functions and check if the corresponding values of bins are all set, is enough. If the result is positive one may deduce that with a small error probability the packet satisfies the rule and can be delivered to the second processing stage for a more in-depth analysis.

In the following, two specific types of efficient data structures are described. The first one is a probabilistic counter, namely the *loglog* counter which proves to efficiently estimate the cardinality of large multi-sets in a very memory efficient manner. The second data structures is a particular type of sketch that can be reversed on–demand. Both data structures will be later adopted in the practical use case given in the chapter 6.

1.6.1 Probabilistic counters

Probabilistic counters are intended as a class of algorithms that estimate the number of distinct elements in a set. Naturally, such an objective could be achieved exactly, at the expenses of a huge memory footprint when the number of elements to count becomes very large. In fact, this is precisely the case of traffic data on high–speed networks, especially in the presence of malicious packet flooding.

Probabilistic algorithms like LogLog counters [43], instead, "trade certainty for time/space" [110] by estimating the number of unique occurrences of elements in large multisets through impressively compact data structures at the expenses of a small error rate.

More formally, given a multiset M produced starting from a discrete universe U, the objective is to estimate the cardinality of the support of M (aka its *dimension*), namely the number of *distinct* elements it comprises. Like in many similar algorithms, even in this case it can be assumed that a hash function $h : U \rightarrow \mathcal{U}$ is available for transforming each element of U into sufficiently long binary strings $x \in \mathcal{U}$ producing a "random" multiset $\mathcal{M} = h(M)$ with n distinct elements. Note that the use of hash functions allows obtaining strings x with random uniform independent bits.

Given that, let us suppose that the strings are infinitely long, that is $\mathcal{M} = \{0,1\}^\infty$ (this is a convenient abstraction at this stage) and let $\rho(x)$ denote the (1 based) position of its first 1-bit. Consider now the set $\mathcal{P} = \{\rho_1, \rho_2, \dots \rho_n\}$, with $\rho_i = \rho(x_i), x_i \in \mathcal{M}$. The elements of $\mathcal{P}$ form a sequence of independent and identically distributed geometric random variables with common pmf:

$$\Pr\{\rho = k\} = \left(\frac{1}{2}\right)^k, k \geq 1 \tag{1.1}$$

Hence, the maximum of the set $\mathcal{P}$:

$$R(M) = \max_{x \in M} \rho(x) = \max_{1 \leq i \leq n} \rho_i \tag{1.2}$$

has the cumulative distribution function:

$$\begin{aligned}
\Pr\{R \leq k\} &= \prod_{i=1}^{n} \Pr\{\rho_i \leq k\} && (1.3)\\
&= (\Pr\{\rho_i \leq k\})^n && (1.4)\\
&= 1 - (\Pr\{\rho_i > k\})^n && (1.5)\\
&= \left(1 - \left(\frac{1}{2}\right)^k\right)^n && (1.6)
\end{aligned}$$

and its mean value [106]:

$$\begin{aligned}
\mathbb{E}[R] &= \sum_{i=1}^{n} \Pr\{\rho_i > k\} && (1.7)\\
&= \sum_{i=1}^{n} 1 - \left(1 - \left(\frac{1}{2}\right)^k\right)^n && (1.8)\\
&\approx \log_2 n && (1.9)
\end{aligned}$$

provides a reasonably rough approximation of $\log_2 n$. In fact, it turns out [43] that the additive bias of R in estimating $\log_2 n$ is about 1.33, while its standard deviation is around 1.87.

To improve the estimate of n, the elements of $\mathcal{M}$ can be divided into m groups (buckets) ($M^{(j)}$ with $j = 1, 2, \cdots, m$) and compute the parameter R on the strings belonging to each bucket. Typically, $m = 2^k$ so that we can use the first k bits of x to represent the binary index of the bucket. For each group, let the *registers* $R^{(j)} = \max_{x \in M^{(j)}} \rho(\tilde{x})$, where $\tilde{x}$ is obtained by discarding the first k bits of the strings $x \in M^{(j)}$ (indeed, the statistics on the position of the first bit set to 1 do not depend on the offset).

Then, the arithmetic mean:

$$\frac{1}{m} \sum_{j=1}^{m} R^{(j)} \tag{1.10}$$

is legitimately expected to approximate $\log_2(n/m)$, plus an additive bias.

Thus, the estimate of n according to the *LogLog* algorithm is:

$$E = \alpha_m m 2^{\frac{1}{m} \sum_{j=1}^{m} R^{(j)}} \tag{1.11}$$

where α_m is the bias correction factor in the asymptotic limit and can be evaluated as follows [43].

$$\alpha_m = \left(\Gamma(-1/m) \frac{2^{-1/m} - 1}{\log 2}\right)^{-m} \tag{1.12}$$

$$\Gamma(s) = \frac{1}{s} \int_0^{\infty} e^{-t} t^s \, \mathrm{d}t \tag{1.13}$$

Notice that the algorithm needs to store m registers (the values $R^{(j)}$ computed over each buckets) each of them having potentially unlimited length. The authors of [43] found that collecting only the $\theta_0 = \lfloor 0.7m \rfloor$ smallest values (*truncation rule*) and limiting their range to the interval $\left[0 \ldots \lceil \log_2 \left(\frac{n}{m}\right) + 3 \rceil\right]$ (thus limiting their memory occupancy to $\lceil \log_2 \lceil \log_2 \left(\frac{n}{m}\right) + 3 \rceil \rceil$ bits) yields an estimation error for the LogLog counter of $\frac{1.05}{\sqrt{m}}$. They name such an optimized version of this counter *Super–LogLogcounter*.

A further decrease of the estimation error up to $1.04/\sqrt{m}$ has been achieved in [48] by introducing an other optimization named *Hyper–LogLog counter*. Roughly speaking, the performance improvement has been obtained by replacing the arithmetic mean of the LogLog counter with the geometric mean, namely:

$$E = \frac{\alpha_m m2}{\sum_{j=1}^{m} 2^{-R^{(j)}}} \tag{1.14}$$

where α_m is given by:

$$\alpha_m = \left(m \int_0^\infty \left(\log_2 \left(\frac{2+x}{1+x} \right) \right)^m dx \right)^{-1} \tag{1.15}$$

More precisely [48], if $\sigma \approx 1.04/sqrtm$ corresponds to the standard error, the estimates provided by the Hyper–LogLog counter are expected to be within $\pm\sigma$, $\pm 2\sigma$, and $\pm 3\sigma$ of the exact count in 65%, 95%, and 99% of all cases, respectively.

As a final note, it is worth to elaborate upon the memory footprint of such probabilistic counters. Indeed, the main benefit of their use is represented by their extremely low memory occupancy which, in many cases, allows their placement into a reasonably small data structure to reside in small memories, such as fast caches or those on board of IoT devices. The overall memory occupancy of a counter depends on the cardinality of the support of the multiset to estimate as well as on the required estimation accuracy. Indeed, by construction, the memory footprint of any single LogLog counter is that of m registers (also called *small bytes*), each of them sized $\log_2 \log_2 N$ bits, where N is an a–priori estimate of the multiset dimension. Depending on the application, by adequately tuning the parameter m may provide a very compact counter and enable a significant computation speedup.

1.6.2 Reversible Sketches

Sketches have proven to be useful in many data stream computation applications [35], [36], [49]. Recent work on a variant of the sketch, namely the k-ary sketch, showed how to detect large changes in massive data streams with small memory consumption, constant update/query complexity, and provably accurate estimation guarantees [66].

In a nutshell, a sketch (see Fig. 1.6) is a two-dimensional $d \times w$ array $S[l][j]$, where each row l ($l = 1, \ldots, d$) is associated with a given hash function h_l. These functions give an output in the interval $(1, \ldots, w)$ and these outputs are associated with the columns of the array. As an example, the bucket $S[l][j]$ is associated with the output value j of the hash function l.

Considering the input data as a stream that arrives sequentially, item by item, where each item consists of a *hash key*, $i_t \in (1, \ldots, N)$, and a *weight*, c_t, when new data arrive, the sketch is updated as follows:

$$S[l][h_l(i_t)] \leftarrow S[l][h_l(i_t)] + c_t \tag{1.16}$$

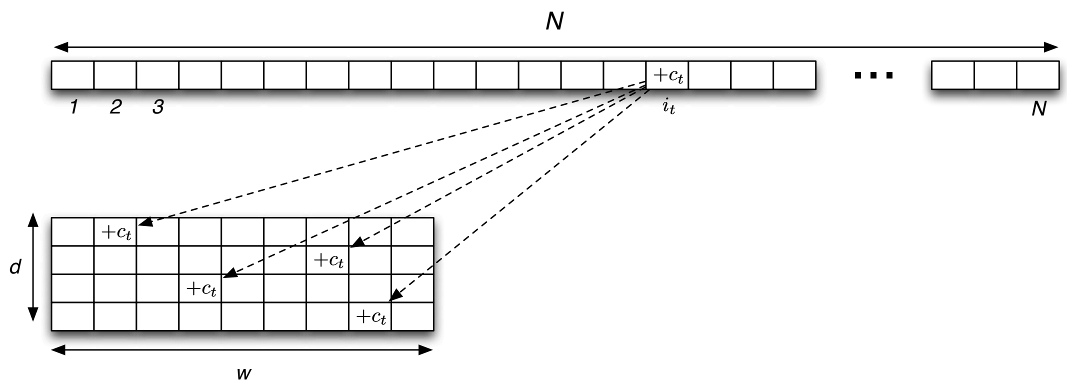


Figure 1.6: *k-ary Sketch*

The update procedure is realized for all the different hash functions as shown in figure 1.6.

However, sketch data structures have a major drawback: they are usually not reversible. In other words, a sketch cannot efficiently report the set of all keys that correspond to a given bucket.

Such limitation can be removed by modifying the input keys and hashing functions to make it possible to recover the keys with certain properties without sacrificing the detection accuracy. With this approach, the paper [95] proposes a novel algorithm for efficiently reversing sketches, by modifying the update procedure for the k-ary sketch through modular hashing and IP mangling techniques.

The modular hashing operates by first partitioning the n-bit long hash key x into q words $x_1, x_2, \ldots, x_q$ of equal length n/q such that $x = x_1|x_2|\ldots|x_q$. Each word is then hashed separately by using a different hash function, h_{di} ($i = 1, \ldots, q$), to obtain an m-bit long output. Finally, these outputs are concatenated to form the final hash value (see Fig. 1.7):

$$\delta_d(x) = h_{d1}(x_1)|h_{d2}(x_2)|\ldots|h_{dq}(x_q) \tag{1.17}$$

Hence, the final hash value consists of $q \times m$ bits which, in turn, makes the number of column of the sketch equal to $w = 2^{q \times m}$.

Note that the use of the modular hashing may cause a highly skewed distribution of the hash outputs. Consider, as an example, the widely typical case in which IP addresses are used as hash keys. In network traffic streams there are strong spatial localities in the IP addresses since many IP addresses share the same prefix. As a result, the first octets (equal in most addresses) will be mapped to the same hash values increasing the collision probability of such addresses.

To efficiently resolve this problem, the *IP mangling* technique has to be applied before computing the hash functions. By using such a technique, the system randomizes, in a reversible way, the input data to remove the correlation or spatial locality.

Reversing algorithm. The full description of the algorithm for reversing the sketch is given in [95]. In a few words, the rationale of the algorithm is to check separately all possible words of the keys to find a match with the corresponding portion of bucket address in the sketch. However, the use of hash modularity allows to immediately

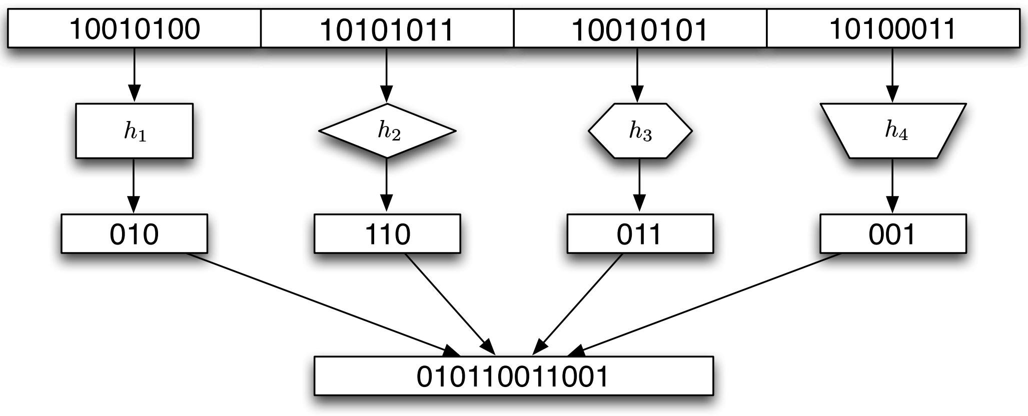


Figure 1.7: *Modular Hashing*

prune out of the whole cartesian products of words all of the q-uples in which even a single word does not hit any sub–address of the buckets to invert. This way the inversion algorithm converges very quickly and produce *all of the keys that may hit to the selected buckets* in the sketches. Note, however, that the output of the algorithm only depends on the "geometry" of the sketch, namely the bitwise length of the keys and on the specific hash functions used to populate it. In fact, a subset of the keys returned by the algorithm may not even be present in the sketch and represents false positive. A further confirmation stage is therefore needed to check *which keys* have been inserted in the sketch. This stage can be efficiently implemented by inserting the keys into a simple *footprint* Bloom filter at the same time they are inserted in the sketch.

1.7 Final remarks

The purpose of this first chapter is to introduce the reader to the practice of high-performance programming for network applications.

For no reason, the chapter aims at presenting the solution to all the problems a network programmer can face. Some aspects have been left out intentionally as they do not directly affect the performance itself. Others, which deserve more attention, have been postponed to the chapters that follow.

It is evident that high-performance network programming requires an in-depth knowledge of the hardware and the low-level mechanisms that govern it. Performance is only achieved through an in-depth analysis of all these aspects.

At the kernel level, for example, the development is very complicated, kernel modules are difficult to maintain (given the rapid variability of the APIs) and very annoying to debug – there is no practical way to look for a bug without running into a kernel panic.

Huge attention must be paid to concurrent and lock-free programming. These techniques necessarily require the use of assembly code or some extensions recently introduced in the C and C++ languages to shape and control the way writes to memories propagate to cores (Memory Models). This semantics is hard to implement and is equally complicated to finalize, as there are no tools that allow verifying the presence of a race-condition. This fact, in turn, renders the execution of the application non-deterministic, or in other words, non deterministically reproducible.

It is straightforward to conclude that this kind of applications requires the adoption of too much knowledge that, in turn, forces the developer to focus his attention on the gory details rather than on what he is indeed in charge to design.

Therefore, the experience we matured in the recent years has led us to create a framework that hides the most complicated details to the programmer, providing high-level APIs that can be used to implement fast and robust network applications. Such a framework is presented in the following chapter.

CHAPTER 2

Fast Packet I/O

High-speed data availability and efficient data processing call for (at least) two complementary – and somewhat orthogonal – features for a network device in charge of running one of the above-listed network applications: high-speed traffic capturing and effective dispatching mechanisms to upper-level applications.

From a technological point of view, in the last years the evolution of commodity hardware has been pushing parallelism forward as the key factor to allow software-based solutions to attain hardware-class performance while still retaining its advantages. Indeed, on one side commodity CPUs provide more and more cores, while on the other side a new generation of NICs support multiple hardware queues that allow cores to fetch packets concurrently. For these reasons, commodity PCs have recently become increasingly popular to be the underlying hardware platforms for the development of complex and high-performance network applications, switches, middleboxes and so on.

As CPU speed has nearly reached saturation values, parallel processing emerges as the natural way to let network applications scale up to multi-gigabit (10/20/40 Gbps) line speed. Indeed, almost any non-trivial monitoring application in charge of operations like reconstructing TCP flows, computing statistics, performing DPI and protocol classification, etc. requires at least a few thousands of clock cycles per each observed packet. In most cases, a large amount of clock cycles is wasted in accessing the data structures that contain the state of the flows under investigation rather than in packet elaboration itself. This way, even a simple application that consumes around 3000 clock cycles per packet cannot process more than 1 million of packets per second on a 3 GHz core. In all such cases, distributing the workload to multiple cores is the only viable approach to improve speed performance and maintain the application usability.

In this chapter, we present a general purpose framework named PFQ for high-speed packet capturing and distribution to network devices and applications (endpoints) running on Linux based commodity PCs. The primary objective of PFQ is to handle the application parallelism by allowing fine-grained configuration of packet dispatching from the capture interface to user-space processing applications.

The PFQ project [20] started a few years ago and was born as a Linux kernel capture engine [21]. Since then, the platform has changed quite a lot, and many features have been added, including an in-kernel programmable stage for early processing [23]. In its current shape, PFQ enhances network I/O capabilities and enables easy configuration for user-space parallel processing while preserving the host system normal behavior (including device drivers and kernel data structures). As such, PFQ masquerades the low-level capture complexity and exposes a set of processing abstractions to new multi-threaded applications or legacy single-process programs.

2.1 Background and Motivations

The investigation on software-based approaches to traffic capturing, monitoring and – more generally – processing running on commodity PCs has recently emerged as an appealing topic in the research community as a viable and cheap alternative to traditional hardware solutions. At the lowest level, to overcome the performance limitations of a general purpose operating system, many techniques have been proposed to accelerate packet capturing. Most of them rely on bypassing the entire operating system, or at least its network stack functions. An extensive comparison of such techniques along with guidelines and possible improvements to reach higher performance can be found in [26] and more recently in [77] and [51].

PF_RING [50] was one of the first software accelerated engines. It uses a memory mapped ring to export packets to user space processes: such a ring can be filled by a regular sniffer or by modified drivers, which skip the default kernel processing chain. PF_RING works with both vanilla and aware drivers, although its performance makes it more suitable for 1 Gbps links. PF_RING ZC (Zero Copy)[1] [39] and Netmap [93], instead, memory map the ring descriptors of NICs at user space, allowing even a single CPU to receive 64 bytes long packets up to full 10 Gbps line speed. A step forward to network programming is represented by DPDK [42] that, besides accelerating traffic capture through OS bypassing, adds a set of libraries for fast packet processing on multicore architectures for Linux. OpenOnLoad [103] provides a high-performance network stack to accelerate existing applications transparently. However, its use is strictly limited to SolarFlare products.

HPCAP [78] is a packet capture engine designed to optimize incoming traffic storage into non-volatile devices and to provide timestamping and user-space delivery to multiple listeners.

Out of the above-listed frameworks, DPDK, PF_RING ZC, and Netmap hit the best performance in capturing and bringing packets to user-space applications at multi-gigabit line rates, even with a single capturing CPU.

At a logically higher level, many interesting works have been carried out for designing software-based switches and routers: although their scope is different, several common grounds with network monitoring are easily found, the most important being

[1]The successor of the formerly known PF_RING DNA

the need for de-queuing packets at wire speed.

Packetshader [55] is a high performing software router that takes advantage of GPU power to accelerate computation/memory intensive functions. Egi *e*t al. [44] investigate on how to build high-performance software routers by distributing workload across cores while Routebricks [40] proposes an architecture to improve the performance of software-based routing by using multiple paths both within the same node and across multiple nodes forming a routing cluster.

The last two works rely on the Click modular router [65], a widely known framework that allows building a router by connecting graphs of elements. Several works have recently focused on the acceleration of Clicks using some of the above listed I/O frameworks as in [92] and [13]. Furthermore, the Click approach has recently been complemented to take advantage of GPU computational power in Snap [105]. The Click modular principle is borrowed by Blockmon [57], a monitoring framework which introduces the concept of primitive composition on a message passing based architecture. Finally, the Snabb switch [102] combines the kernel bypass mode of Ethernet I/O with the use of the Lua scripting language to build a fast and easy to use networking toolkit.

So, why PFQ?

The introduction of a new generation of network cards with multiple hardware queues has pushed a significant evolution of existing I/O frameworks to support Receive Side Scale (RSS) [59] technology. As it will be elaborated upon in the following, RSS uses the Toeplitz hashing mechanism to split the incoming traffic across multiple hardware queues for parallel processing. The hash is computed by the network card itself over the canonical 5-tuple of IP packets and traffic is spread out among cores maintaining per-core uni-directional flow coherency by default. Besides, the hash algorithm can be properly tweaked to achieve bi-directional flow coherency [113]. However, in many real cases, a more refined distribution criteria is required by applications and multi-core processing management merely based on RSS turns out to be insufficient. As a simple example, to run multiple instances of the well known NIDS application Snort [33], a special symmetric hash function to achieve network-level coherency is required to detect cross-flow anomalies and attacks within a specific LAN properly.

Slightly more complex examples include service monitoring applications that require either data and control plane packets to be processed by the same thread/process (e.g. RTP and RTCP or SIP) or tunneled protocols (IPIP, GRE, GTP), whereas RSS would spread traffic according to the tunnel headers instead of the inner packets fields.

PF_RING supports packets distribution through the commercial *PF_RING ZC library*[2]. According to the documentation [82], the library is equipped with algorithms for dispatching packets across endpoints. Such functions take an extra user-defined callback to fully specify the balancing behavior (by a hashing scheme). Also, to ease the implementation of such callbacks, the library provides helper functions that compute a symmetric hash on top of IP packets, possibly transported by GTP tunnels.

Similarly, the companion *Distributor library* of DPDK [41] implements a dynamic load balancing scheme. The module takes advantage of the RSS tag stored in the `mbuf` structure to sequence and dispatch packets across multiple workers.

Both the above solutions are designed to embed a packet distribution machinery into

[2]The evolution of the formerly known libzero library

applications to implement multi-threaded packet processing. However, such solutions are not entirely transparent to the applications which, in turn, need to be adapted to take full advantage of these mechanisms.

As previously mentioned, the PFQ project started in 2011 and appeared first in [21]. Since its original version, PFQ was designed as a software capture engine with a basic in-kernel steering stage targeted to allow user-space applications defining their arbitrary degree of parallelism. Nevertheless, the initial version of PFQ required modified network device drivers to reach high performance.

The current version of PFQ, instead, is compatible with a wide plethora of network devices as it only requires the original vanilla drivers to be recompiled with a script included in the package to achieve full acceleration. However, PFQ can work with *binary* vanilla drivers as well, although I/O performance may drop depending on some factors, including driver and kernel versions. As an example, the use of the binary 10G *ixgbe* Intel driver shipped with Linux kernel 3.16 allows hitting slightly less than half of the optimal capturing rate.

Generally speaking, the use of existing vanilla drivers might result in limited performance figures whenever their quality is not adequate. However, nothing prevents PFQ from using modified/optimized drivers to further boost performance.

Also, PFQ is equipped with an in-kernel processing stage programmable through the functional language pfq-lang, available as an embedded Domain Specific Language (eDSL) for the C++ and the Haskell languages. To further improve usability, an experimental compiler also allows pfq-lang instructions to be *scriptable* and placed in strings, configuration files, and JSON descriptions. As a result, no programming skill is needed to accelerate legacy applications as they do not require any modifications.

The pfq-lang language is *extensible* and *pluggable*. Additional in-kernel functions can be added to the language in separated kernel modules and loaded on the fly as plugins. Moreover, pfq-lang computations are dynamic and hot-swappable (i.e., run-time atomically upgradable) to be used, for instance, in response to either network events or configuration updates. As a result, As a whole, up to 64 multi-threaded applications can be bound to the same network device, each of them receiving an independent and fully configurable quota of the overall underlying traffic.

This chapter aims at providing a complete overview of PFQ by adding a detailed description of its architecture and its software acceleration internals to the functional engine described in [23] and therein assessed *in isolation* only. Besides, a set of practical use-cases is presented to show how PFQ can be used in practice and to evaluate its effectiveness in accelerating new and legacy applications.

2.2 The System Architecture

The architecture of PFQ as a whole is shown in Figure 2.1. In a nutshell, PFQ is a Linux kernel module designed to retrieve packets from *one or more* traffic *sources*, make some elaborations utilizing functional blocks (the λ_i blocks in the picture) and finally deliver them to one or more *endpoints*.

Traffic sources (at the bottom end of the figure) are represented by Network Interface Cards (NICs) or – in case of multi-queue cards – by single hardware queues of network devices.

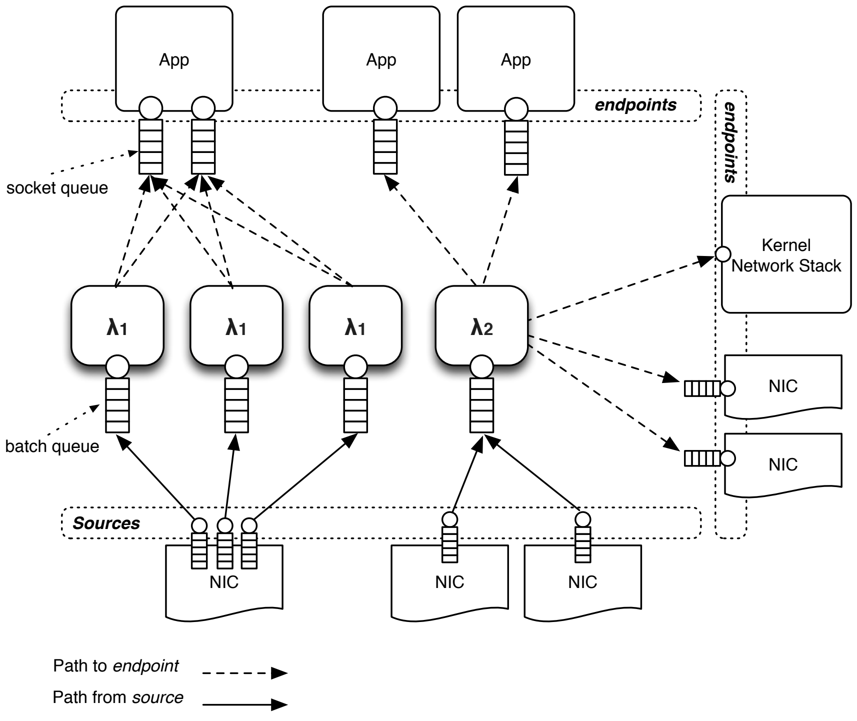


Figure 2.1: *The PFQ system at-a-glance*

Endpoints, instead, can be either sockets from user-space applications (top end of the figure) or network devices, or even the networking stack of the operating system itself (right end of the figure) for ordinary operations (e.g., traditional routing, switching, etc.).

The low-level management of NICs is left under the OS control as the network device drivers are not modified.

A typical scenario for the use of PFQ is that of a multi-threaded network application in charge of monitoring traffic over one or more network interfaces. Each thread opens a single socket to the group of devices to monitor and receives a quota (or all) of the packets tapped from the selected interfaces. On their way to the application, packets come across functional blocks, that may implement part of the application processing machinery. The execution of such an early stage of elaboration is instantiated by the application itself through a functional eDSL. As a result, packets are finally delivered to the selected endpoints.

It is worth noticing that PFQ does not bypass the Linux kernel, but stands merely along with it. Packets directed to networking applications are treated by PFQ exclusively, whereas packets destined for other system operations can be transparently passed to the kernel stack, on a per-packet basis.

Figure 2.2 depicts the complete software stack of the PFQ package. The kernel module includes a reusable pool of socket buffers and the implementation of a functional language along with the related processing engine. In the user-space, the stack includes native libraries for C++11-14 and C language, as well as bindings for Haskell language, the accelerated pcap library and two implementations of the eDSL for both C++ and

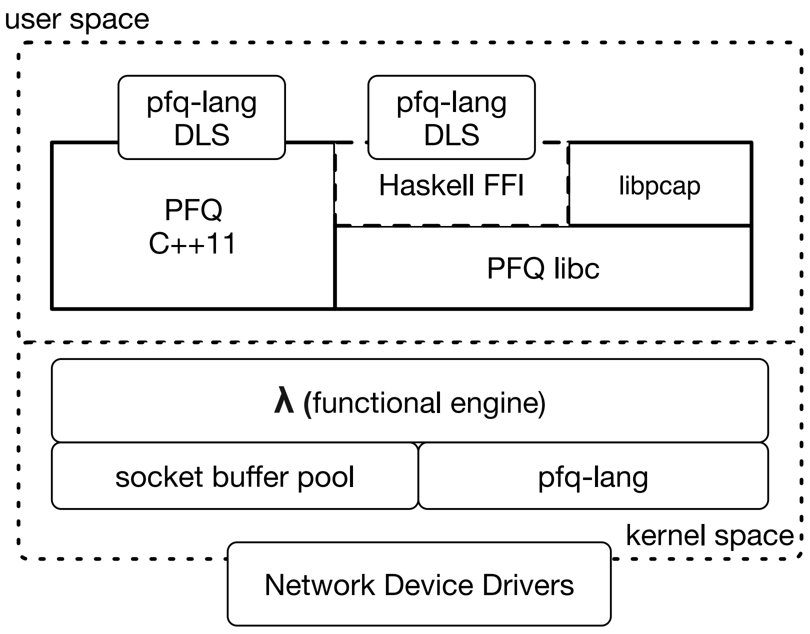


Figure 2.2: *PFQ software stack*

Haskell.

2.2.1 Three layers of parallelism

The system architecture depicted in Figure 2.1 reveals three distinct levels of parallelism associated with three different areas:

- at the low (hardware) level, where packets can be retrieved from multiple NICs and multiple queues of modern network cards;
- at the top level, where multi-threaded applications or multiple single-threaded applications may want to process packets with a different degree of parallelism;
- at the middle level, where multiple kernel threads running on different cores tap packets from network cards and serve user-space applications, network cards or the network stack of the OS.

The above levels of parallelism may additionally get combined in several possible schemes, as shown in Figure 2.3 where endpoints (E), functional engines (λ) and sources (S) can be configured according to different degrees of parallelism.

Hardware parallelism. At the hardware level, modern NICs (such as those based on the Intel 82599, X520 or X710 controller) support multiple queues: multiple cores can therefore receive and transmit packets in parallel. In particular, incoming packets are demultiplexed by RSS technology [59] to spread traffic among receiving queues using a hash function computed over a configurable number of packet fields. Each queue can be bound to a different core by properly setting its *interrupt affinity* to balance the overall capture load among the computation resources of the system.

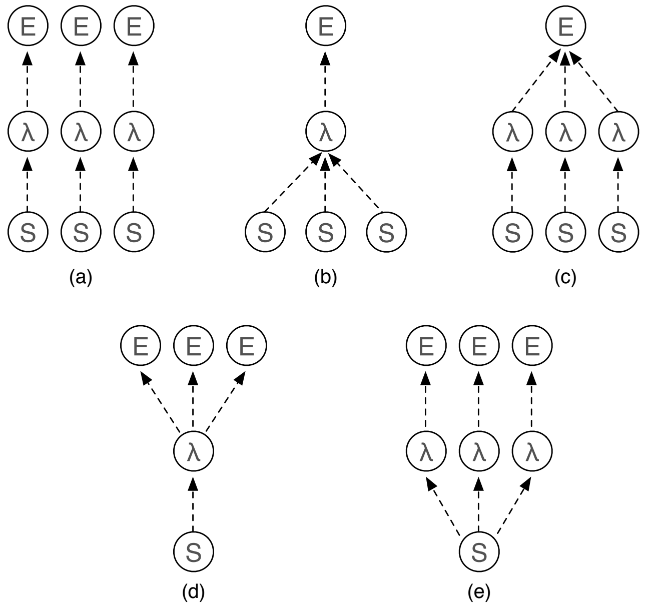


Figure 2.3: *Three layers of parallelism*

In-kernel parallelism. Once the interrupt affinity has been set and the capture parallelism is enabled, each CPU is in charge of processing a quota of the incoming traffic according to the RSS algorithm.

Here is where PFQ intervenes with its operations. PFQ runs on top of *ksoftirqd* threads that retrieve packets in parallel from network device upon receiving an interrupt. NAPI is used to mitigate interrupt rate as in a standard Linux kernel. Hence, the number of kernel threads in use, say i, equals the number of NAPI contexts enabled on the system and throughout the chapter will be referred to as $RSS = i$ (instead, when no further specification is given, the term thread alone refers to an application thread of execution).

The combined use of RSS and interrupt affinity allows a fine-grained selection of the NAPI kernel threads and the hardware queues of network cards. At this stage, packet payloads are transferred via DMA from the wire to RAM, and the Intel DCA mechanism makes them available to the CPU cache memory with no need for extra memory accesses.

Packets handled by PFQ are then processed through the functional engines and optionally steered to endpoints according to application-specific criteria. Upon steering, however, cache locality cannot be preserved as packet payloads might be transferred to different cores. This is the necessary price to pay to let user-space applications/threads

Overall, hardware parallelism turns out to be decoupled from the user-space applications which, in turn, only see the PFQ sockets and the related APIs exposed by the companion libraries.

The advantage of an in-kernel built-in engine is two-fold: on the one hand, it allows fine-grained control over the distribution process before packets are delivered to sockets without extra packet copies. On the other, it is entirely transparent to applications (including legacy ones) which, in turn, do not require any modification to perform a proper parallel processing. Also, the steering process brings up the kernel and network devices along with sockets, which makes it suitable not only for traffic monitoring but also for more advanced networking applications such as packet brokers, load-balancers, etc.

Application parallelism. At the top level, network applications (or more generally, endpoints) are allowed to receive traffic from one or more network devices according to different schemes. As already mentioned, the typical scenario is that of a multi-threaded application in which each thread receives a portion of traffic from one or more network devices. But one may also think of multiple process instances (typically, legacy applications) that run in parallel and receive a portion of the underlying captured traffic as well. Or even a single process collecting all of the traffic from multiple network devices.

As will be elaborated upon in a few sections, all such cases are flexibly handled by PFQ through the abstractions of *groups* and *classes* that provide convenient extensions to the concept of network socket in case of parallel processing. Applications will only need to register their sockets to specific groups, without any knowledge about the underlying configuration.

Also, multiple logical schemes from Figure 2.3 may be instantiated at the same time as different applications may run concurrently to process traffic from the same set of network devices. PFQ can perfectly cope with this scenario as different applications will use distinct groups that run orthogonally to each other making any application behave just as it were the only one running on the system.

2.3 High Speed Packet Capture

This section describes the PFQ internals associated with the operations involved in the low level packet capture. As introduced above, the design philosophy of PFQ is to avoid modifications to the network device drivers and their interfaces toward the operating system.

If on one side the use of vanilla drivers allows a complete compatibility with a large plethora of network devices, on the other side it could raise performance issues. Indeed, the standard OS handling of packet capture cannot guarantee decent performance on high-speed links and successful projects like PF_RING ZC or Netmap have demonstrated the effectiveness of driver modifications.

However, with the software acceleration techniques implemented in PFQ, it is still possible to achieve top class capture figures while retaining full compliance with standard driver data structures and operations. The impact of such acceleration techniques will be thoroughly assessed in the corresponding section of the performance evaluation results.

2.3.1 Accelerating vanilla drivers

The first performance acceleration technique introduced by PFQ consists of intercepting and replacing the OS functions invoked by the device driver with accelerated routines. This way, the kernel operations triggered by the arrival of a packet are bypassed, and the packet itself gets under the control of PFQ. This procedure does not require any modification to the source code of NIC drivers which, in turn, only need

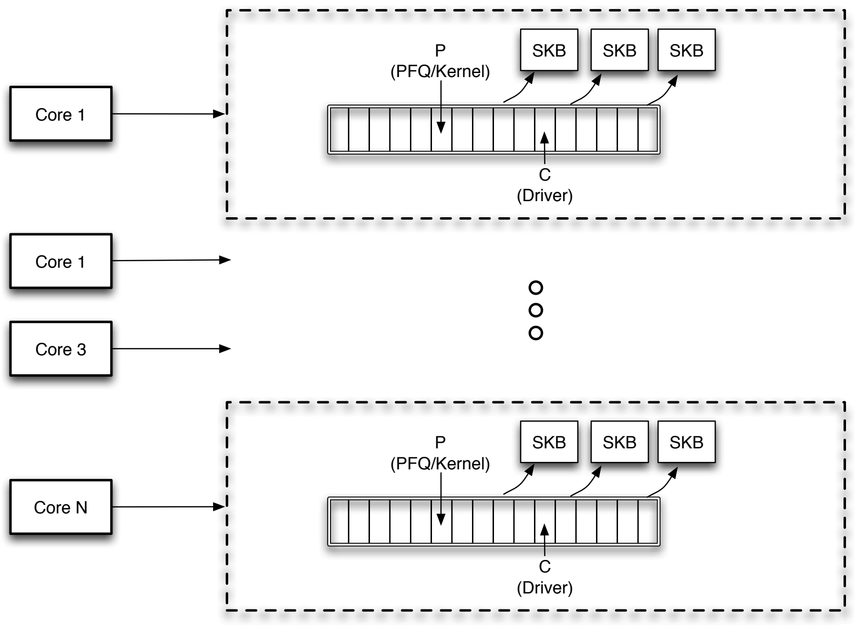


Figure 2.4: *Pool of skbuffs*

to be compiled against a PFQ header to *overload* at compile time the relevant system-calls that i) pass packets to kernel (namely `netif_receive_skb`, `netif_rx` and `napi_gro_receive`) and ii) are in charge of allocating memory for the packet (e.g., `netdev_alloc_skb`, `alloc_skb`, `dev_alloc_skb`, etc.).

Such a static function overloading does not introduce any overhead. Also, the whole operation is made easier by the `pfq-omatic` tool included in the PFQ package that automates the compilation and only needs the source code of the vendor device drivers.

2.3.2 Pool of socket buffer

The typical behavior of a network device driver is to allocate a set of *socket buffers (skbuffs)* where the NIC can place (via DMA) the payload of the packets received, together with additional metadata (timestamp, etc.). Once the *skbuffs* are ready, they can then be passed back to the network stack of Linux.

To keep the full compatibility with standard driver operations and to allow a possible delivery to the system OS (when it acts as an endpoint) PFQ maintains this design, while accelerating the *skbuff* memory allocations by making use of pre-allocated pools of *skbuff* (Figure 2.4). Such pools have a configurable maximum size and are instantiated one per-core to avoid inter-core data sharing, as opposed to the case of the standard OS kernel that, instead, implements a single *kmem_cache* of *skbuff* for the whole system. Initially, each pool is empty and the *skbuffs* are allocated on-demand by the device drivers (using the standard memory allocator for *skbuff*). After the completion of their processing, the consumed *skbuffs* are parked in the pool for reuse. After a very short time, each pool contains enough recycled *skbuffs* to be reused upon driver request, so that the kernel allocator is no longer needed.

In queueing terminology, the pool can be modeled as a circular single producer/single consumer queue, in which producer and consumer run on the same core.

As a special case, it is relevant to note that packets forwarded to the kernel

2.3.3 Batch queues

Once an *skbuff* is received by PFQ it is first placed in a *batch queue*. PFQ maintains one batch queue per (active) core. Mainly, these queues are standard FIFO used to place packets before they are processed in batches by the functional engines (when the queue is full or when a timeout expires). Batch processing has demonstrated to be a very effective acceleration technique for at least two reasons. The first reason is that batching operations always improve the temporal locality of memory accesses that, in turn, reduce the probability of cache misses. However, the primary effect is determined by the dramatic *amortization* of the cost of the atomic operations involved in the processing of packets. Indeed, even the simple distribution of packets to sockets requires at least a per-packet atomic operation. The use of batch processing allows decreasing the cost of such an overhead of a factor $1/batch\_size$, with clear performance improvement.

The use of batch processing allows decreasing the cost of such an overhead of a factor $1/batch\_size$, with clear performance improvement.

The size of the batch queues is configurable as a PFQ module parameter. The impact of the batch queue and its length will be next presented and discussed in the performance section.

2.4 Functional processing

Packets backlogged in the batch queues wait their turn to be processed by functional engines. Each functional engine runs up to 64 distinct *computations* instantiated by upstream applications through a functional language. Computations represent the compositions of *primitive functions* that take an *skbuff* as input and return the *skbuff* possibly enriched with a context specifying an *action*, a *state* and a *log* for I/O operations. *Actions* are associated with the final endpoints and the delivery mode (the packet *fanout*) of the packet. The *state* is associated with *annotations* of metadata on packets while *logs* represent information associated with the packet that is possibly used to generate I/O.

In the functional world, such primitives are named *monadic functions* and their composition is known as *Kleisli composition*; a more formal description of the algebra of PFQ computations is provided in [23]. Besides, traditional functions such as predicates, combinators, properties, and comparators are also available.

All computations instantiated on a functional engine are executed sequentially on each packet and in parallel with respect to the other instances of the same computations running on other cores. However, since the functional paradigm does explicitly forbid packet mutability, the order of execution of computations on the same core is irrelevant.

Computations are executed at kernel space though they are instantiated at user-space through the specially developed Domain Specific Language named pfq-lang presented in section 2.5.2. The use of computations is specially targeted at offloading upstream applications by providing an early stage of in-kernel processing.

Currently, the PFQ engine integrates about a hundred primitive functions that can be roughly classified as: protocol filters, conditional functions, logging functions, forwarding (to the kernel or NIC) and fanout functions (mainly, steering).

The last category of functions is particularly relevant as it defines which (and how) applications endpoints will receive packets. The next section focuses on this central point of PFQ operations by introducing the concepts of groups and classes.

2.4.1 Groups and classes

One of the key features of PFQ is the high level of granularity that can be specified to define the final endpoints for packet delivery. This is made possible by the introduction of a convenient abstraction to let multi-threaded user-space applications *share and spread* flows of packets.

Indeed, consider a single threaded application that receives packets from one or more devices (in standard Linux, it can be either a specific device or all of the devices installed on the system). Such an operation requires opening a socket and binding it to the involved devices. The socket itself, hence, acts as the software abstraction for the *pipe* where packets are received.

In multi-threaded applications, threads reside on top of multiple cores. In this context, the above abstraction of *pipe* needs to be extended to let all of the threads involved in packet handling receive only a portion of the data flowing in the pipe. To this aim, PFQ introduces the abstraction of *group* of sockets. Under this abstraction, each endpoint (thread or process) opens a socket and *registers* the socket to a group. The group is bound to a set of data sources (physical devices or a specific subset of their hardware queues). Also, each group defines its own (unique) computation; hence, each socket participating the group receives packets processed by the same computation in the functional engine.

In a nutshell, a group can be defined as the set of sockets that share the same computation and the same set of data sources.

Different groups behave orthogonally to each other, that is they can transparently coexist on the same system and implement arbitrarily different parallel schemes. In particular, they can access at the same time any arbitrary data source and process and redirect the full amount of retrieved traffic to the registered applications.

The endpoints participating in the same group receive packets according to the fanout primitives introduced at the end of the previous section. Two basic delivery modes are:

- *Broadcast:* a copy of each packet is sent to all of the sockets of the group;
- *Steering:* packets are delivered to the group of sockets by using a hash-based load balancing algorithm. Both the algorithm and the hash keys are defined by the application through the computation instantiated in the functional engine. For example, the function `steer_flow` spreads traffic according to a symmetric hash that preserves the coherency of bi-directional flows, while the function `steer_ip` steers traffic according to a hash function that uses source and destination IP address fields as mega-flows.

Although the concept of group and its delivery modes allow a significant flexibility to the design of user-space applications, it turns out that in many practical cases they are not sufficient to cover the fine-grained requirements of many real network applications.

As an example, consider Figure 2.5 where a multi-threaded application is monitoring the traffic of an arbitrary service from the two network cards reported at the bottom. The application has reserved the special thread shown in the top right-hand side to receive the service control plane packets only, while the remaining threads on the left-hand side are devoted to processing data plane packets. All threads are registered to the same group i, but none of the delivery modes previously described allows to separate traffic according to the application requirements.

The concept of *classes* allows to overcome the problem and increases the granularity of the delivery modes in an elegant way. Indeed, *classes* are defined as a subset of sockets of the group (in fact, a *subgroup*) that receives specific traffic as a result of in-kernel computations. Again, sockets belonging to the same class may receive traffic either in broadcast (here called *Deliver*) mode or in load balancing (here called *Dispatch*) mode.

Coming back to the example of Figure 2.5, it comes out clearly that the combined use of groups/classes and the functional computation easily allow fulfilling the application requirements. Indeed, traffic captured from the network devices are filtered at the functional engines: at this stage, data plane packets are sent in steering mode to the threads belonging to Class 1 (in charge of collecting data plane packets) while control plane packets are sent to the thread belonging to Class 2 (notice that, in this specific case, Deliver and Dispatch mode are obviously equivalent).

Once again, Figure 2.5 evidences the total decoupling between application level and hardware level parallelism allowed by PFQ in which user-space threads join the group/class and receive traffic according to their need without any knowledge about the underlying configuration of hardware devices and the parallel scheme implemented at the kernel level.

The maximum number of groups and classes allowed by the PFQ architecture is 64 for both. Other practical examples of the use of groups and classes will be provided in the use-cases section 2.8.

Groups access policy. Although the concept of the group allows different sockets to participate and share common monitoring operations, for security and privacy reasons not all processes must be able to freely access any active group To this aim, PFQ implements three different group access policies:

- *private*: the group can only be joined by the socket that created the group;
- *restricted*: the group can be joined by sockets that belong to the process that created the group (hence the group is open to all threads of execution of the process);
- *shared*: the group is publicly joinable by any active socket on the system.

2.5 User to Kernel space communication and APIs

This section describes how packets distributed from the in-kernel functional engines reach user-space endpoints and how user-space applications can take advantage of the flexibility provided by the underlying PFQ computation machinery. As such, the internal software mechanisms that implement communication between PFQ and user applications are reported below. Next, the focus of the discussion will turn to the set of application programming interfaces exposed by PFQ to build network applications.

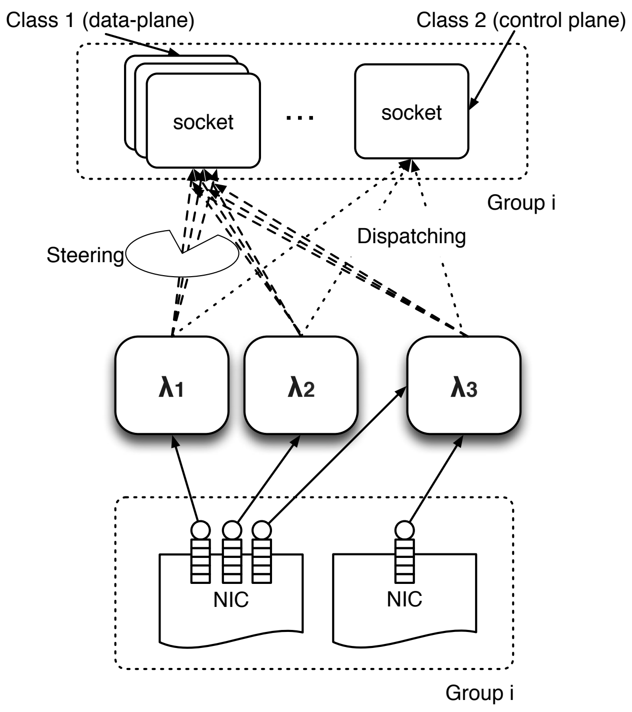


Figure 2.5: *PFQ functional processing and fanout*

2.5.1 User to Kernel space communication

Packets delivered to user-space sockets are placed on a shared memory between user and kernel spaces where special multiple producers/single consumers lock-free queues are allocated. The producers of the queues are the kernel threads running the functional engines while the (single) consumers are the user-space application threads. Each user-space socket consumes packets from its queue; later, they will be referred to as *socket queues*. In modern Linux systems, such queues are allocated on 2 MB large *hugepages* [69]; when this is not possible, PFQ automatically rolls back to standard system pages of 4 KB size.

Socket queues (Figure 2.6) are equipped with a double buffer. The use of such a dual buffer allows decoupling the operations of producers and consumers. While the producers fill one buffer with batches of packets, packets from the other buffer are consumed by the user application. Each time the consumer has exhausted the packets, it triggers the *atomic swap* of the buffer and starts pulling packets from the other buffer.

The atomic swap is triggered upon the (atomic) replacement of the index that identifies the active producer buffer together with a read and immediate reset of the counter of packets placed in the buffer by producers. This atomic swap is made possible on any system in that the buffer index and the packet counters are *both* contained on the same 32-bit integer, in the higher and lower parts, respectively.

As a final remark, note that the socket queue does not prevent packet losses. Indeed, whenever the consumer becomes slower than the producer, the buffer overflow may occur, and packets are dropped.

Egress sockets. A specific discussion is needed when the final endpoint is not a

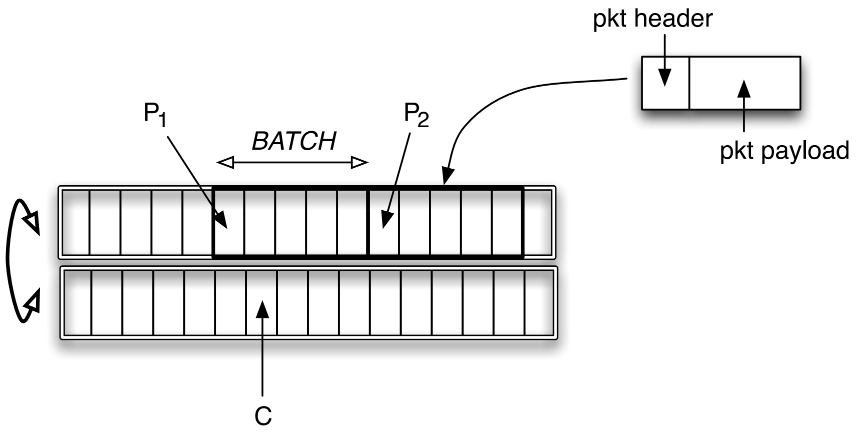


Figure 2.6: *Double buffered socket queue*

socket, but rather a network device. In such case, the socket queue is not used, and the communications are implemented differently. To this aim, PFQ implements a proper abstraction called *egress socket* that adapts the PFQ interface towards generic endpoints with no need to change the distribution operations.

2.5.2 Application Programming Interfaces

From the application programmer point of view, PFQ is a polyglot framework that exposes *native libraries* for the C and C++11-14 languages, as well as bindings for Haskell.

Moreover, beside the traditional APIs, PFQ additionally includes a Domain Specific Language (pfq-lang) that allows programming the kernel-space computations from both embeddable expressions (C++ and Haskell) and configuration files, using its internal compiler. Finally, for compatibility with a large number of traditional legacy applications, PFQ also exposes an adaptation interface towards the standard pcap library.

Native APIs. Native PFQ libraries include a rich set of functions to control the underlying PFQ mechanisms, to handle traffic capture/transmission, to retrieve statistics and to inject in-kernel pfq-lang computations.

It is worth pointing out that the injection of the computations occurs once its formal correctness has been validated at compile time by the C++/Haskell compilers, or by the pfq-lang compiler itself. Additionally, the correctness of the computations is checked again by the kernel itself before being enabled for execution.

pfq-lang. The packet processing pipeline (computation) executed by functional engines can be described by composing multiple functions implementing elementary operations. pfq-lang provides a rich set of functions and is designed to be extensible; this allows users to add functions for their specific purposes easily. Like any functional language, pfq-lang supports high—order functions (functions that take or return other functions as arguments) and currying, that convert functions that take multiple arguments into functions that take a single argument. Besides, the language includes conditional functions and predicates to implement a basic code control flow. Since pfq-lang is used to describe and specify the packet processing logic, its purpose within

PFQ is similar to that of P4 [24] and Pyretic [76] in describing the data plane logic of an SDN network or to that of Streamline [37] to configure I/O paths to applications through the operating system.

As an example, a simple function that filters IP packets and dispatches them to a group of endpoints (e.g., sockets) utilizing a steering algorithm is described as:

```
main = ip >-> steer_ip
```

where `ip` is a filter that drops all the packets but IP ones, and `steer_ip` is a function that performs a symmetric hash with IP source and destination.

pfq-lang implements filters for the most common protocols and several steering functions to serve user-space application requirements. In addition, each filter is complemented with a predicate, whose name begins with `is_` or `has_` by convention.

Conditional functions allow to change the behavior of the computation, depending on a property of the processed packet, as in the following example:

```
main = ip >-> when is_tcp
                   forward "eth1"
                   >-> steer_flow
```

The function drops all non-IP packets, forwards a copy of TCP packets to `eth1`, and then dispatches packets to the group of registered PFQ sockets in steering mode.

The following example shows a simple in-kernel computation for delivering packets by keeping subnet coherence to multiple instances of a Network Intrusion Detection System (e.g., to detect a virus spreading over a LAN):

```
main = steer_net "131.114.0.0" 16 24
```

The network under investigation is specified through its address and prefix (131.114.0.0/16). The second prefix (24) is used as the hash depth to spread packets across the NIDS instances and to preserve class C network coherence.

Libpcap adaptation layer. Legacy applications using pcap library [89] can also be accelerated by using the pcap adaptation layer that has been extended to support PFQ sockets. As an example, the availability of the pcap interface allows multiple instances of single-threaded legacy applications to run in parallel as PFQ *shared* groups can be joined by multiple processes.

However, to keep full compatibility with legacy applications, the pcap adaptation layer is designed to maintain the original semantic and leave the APIs unchanged. Therefore, some specific options needed by PFQ native libraries (such as the ones associated with groups/classes handling, computation instantiations, etc.) are specified as either environment variables or within configuration files.

Pcap acceleration is activated depending on the name of the interface: if it is prefixed by `pfq` the library automatically switches to PFQ sockets, otherwise, it rolls back to traditional PF_PACKET sockets. Also, multiple capturing devices can be specified by interposing the colon symbol (:) between the names of the interfaces (e.g., `pfq:eth0:eth1`).

It is worth noticing that PFQ is transparent to legacy pcap applications running on top of it. As such, for example, they can generally use Berkeley Packet Filters.

In practice, to run on top of PFQ, an arbitrary pcap application such as `tcpdump` should equivalently i) be compiled against the pfq-pcap library or ii) be executed by preloading the pfq-pcap library by means of the `LD_PRELOAD` environment variable.

The following example shows four sessions of `tcpdump` sniffing TCP packets from network interfaces `eth0` and `eth1`. The four sessions run in parallel on group 42 and receive a load-balanced quota of traffic that preserves the flow coherency. The (first) master process sets the group number in use, the pfq-lang computation (steer_flow) and the binding to the network devices. The additional three `tcpdump` instances specify the PFQ_GROUP only, in that all parameters are already set.

```
PFQ_GROUP=42 PFQ_LANG="main = steer_flow"
    tcpdump -n -i pfq:eth0:eth1 tcp

PFQ_GROUP=42 tcpdump -n -i pfq
PFQ_GROUP=42 tcpdump -n -i pfq
PFQ_GROUP=42 tcpdump -n -i pfq
```

2.6 Packet transmission

Although the chapter focuses on the receiving side, it is worth pointing out that PFQ supports packet transmission as well. As such, this section briefly reports on the mechanisms adopted by PFQ for packet transmission.

Roughly speaking, the transmission side of PFQ behaves nearly symmetrically concerning the receiving side.

Socket queues are still double buffered, but the role of producers (the application threads generating traffic) and consumers (the kernel threads in charge of forwarding packets to network devices) is now reverted.

Packets placed into socket buffers by user-space applications are spread out over the different active kernel threads utilizing a hash function that acts as the software dual of the RSS hardware function (named TSS). In turn, PFQ kernel threads fetch packets from socket queues and pass them to the network device drivers for transmission.

As such, the transmission capability of PFQ is mainly used in the experimental sections to feed the PFQ receiving mode with synthetic and real traffic for performance evaluation purposes. In particular, the application `pfq-gen` (included in the PFQ distribution) is used to generate traffic with the desired features (random IP addresses, different packet lengths, etc.) and to replay real traces at different speeds, with randomized (but flow-coherent) IP addresses, and so on.

Also, the packet transmission capability of PFQ is used to effectively accelerate the well known *Ostinato* traffic generator [83]. A detailed report of this practical application is provided in section 2.8.4.

Just as in the receiving side, the transmission mechanisms of PFQ use pure vanilla drivers and take full advantage of bulk network transmission whenever this feature is supported (as in the latest ixgbe and i40e Intel driver versions). Bulk transmission perfectly copes with the PFQ architecture, as the batch mechanism is already present in both the receiving and transmission sides. The experimental investigation reported in the next section evidences the benefits that this feature brings to PFQ performance.

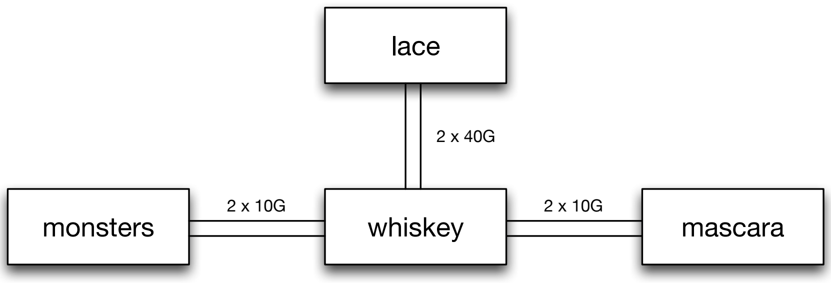


Figure 2.7: *Experimental field trial*

2.7 Performance evaluation

This section aims at assessing the performance of the PFQ architecture under different hardware, kernel and application parallel schemes. Performance of real applications running on top of PFQ will be evaluated separately in the section dedicated to use-cases.

Although PFQ privileges flexibility and usability for bare performance, to be effective it must be able to reach high-speed capturing and processing figures, possibly at the expense of a slight extra cost regarding the amount of system resources needed (i.e., number of cores). The result reported in the following precisely demonstrates that PFQ allows commodity hardware to reach top class performance even by using pure vanilla drivers.

The experimental testbed used throughout the whole set of measurements is shown in Figure 2.7 and is made up of two pairs of identical PCs. Two (older) PCs (*mascara* and *monsters*) with a 6-core Intel Xeon X5650 running at 2.67GHz on board and equipped with an Intel 82599 10G NIC each, used for traffic generation. Two (newer) PCs (*whiskey* and *lace*) with a 8-core Intel Xeon E5-1660V3 on board running at 3.0GHz and equipped with an Intel XL710QDA2 40G NIC each and used for traffic capturing and generation, respectively. Besides, two more Intel 82599 10G NICs were added to whiskey to receive traffic simultaneously generated by mascara and monster on two 10 GB NICs at the same time. All of the systems run a Linux Debian stable distribution with kernel version 3.16.

2.7.1 10G Speed Tests: Packet Transmission

Since PFQ is used to transmit traffic and stress the receiving side, the first set of measurements has the purpose to show that PFQ packet transmission is capable of reaching line rate speed even in the classical worst-case benchmark scenario of 64 bytes long packets. As reported in section 2.6, the user-space application in charge of generating packets and feeding the PFQ transmission engines is `pfq-gen`, an open-source tool included in the PFQ distribution.

Figure 2.8 shows that PFQ reaches the theoretical line transmission rate in all but one case by using a single core (TSS = 1) for transmission. However, line rate performance is achieved even in the case of 64 long byte packet by simply increasing the transmitting kernel threads to 2 (TSS = 2).

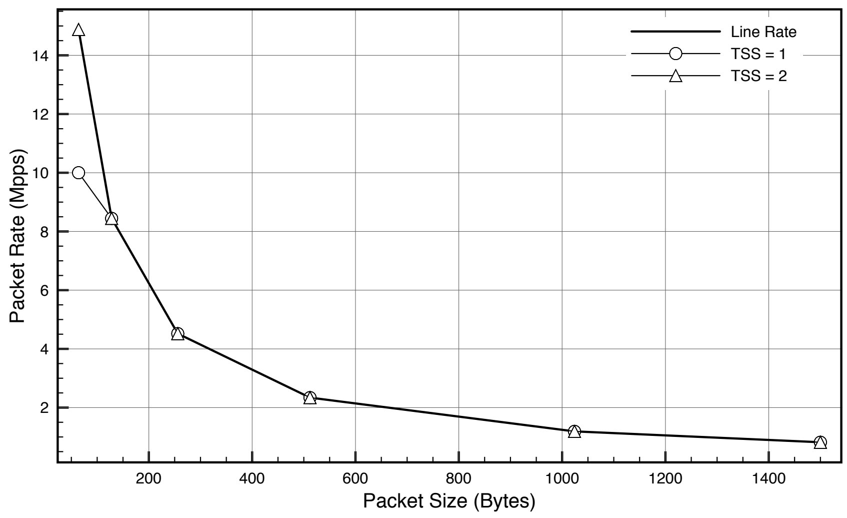


Figure 2.8: *10G packet transmission*

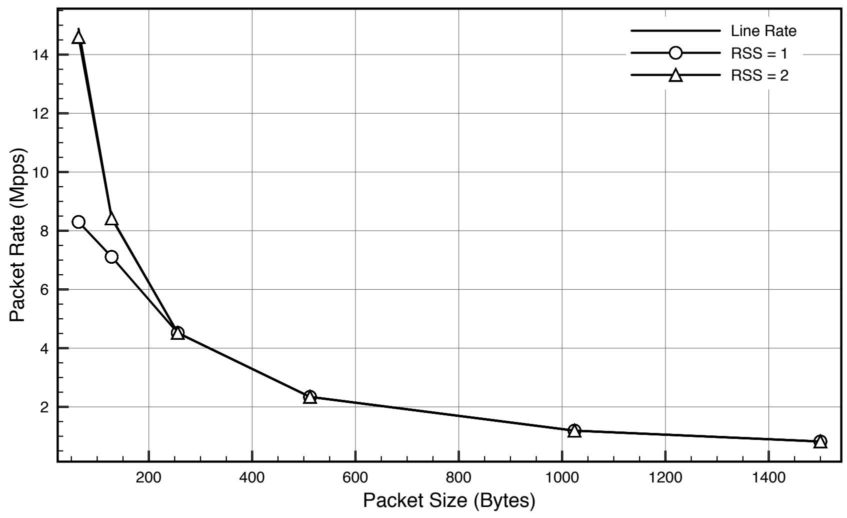


Figure 2.9: *10G packet capture: 1 user space thread*

2.7.2 10G Speed Tests: Packet Capture

The following set of tests aims at checking the pure capturing performance of PFQ under different packet sizes, number of capturing kernel threads and application threads. The user-space application used to receive and compute statistics is `pfq-counters`, a multi-threaded open-source tool included in the PFQ distribution.

Figure 2.9 shows the performance of PFQ when `pfq-counters` uses a single thread to receive traffic from a 10G network interface for different packet sizes and different number of hardware queues (i.e., number of cores used for capture).

In the worst case of 64 bytes long packets, PFQ is capable of handling around 8.3 Mpps per core and, indeed, it requires two kernel engines (RSS=2) to reach line rate performance for all packet sizes.

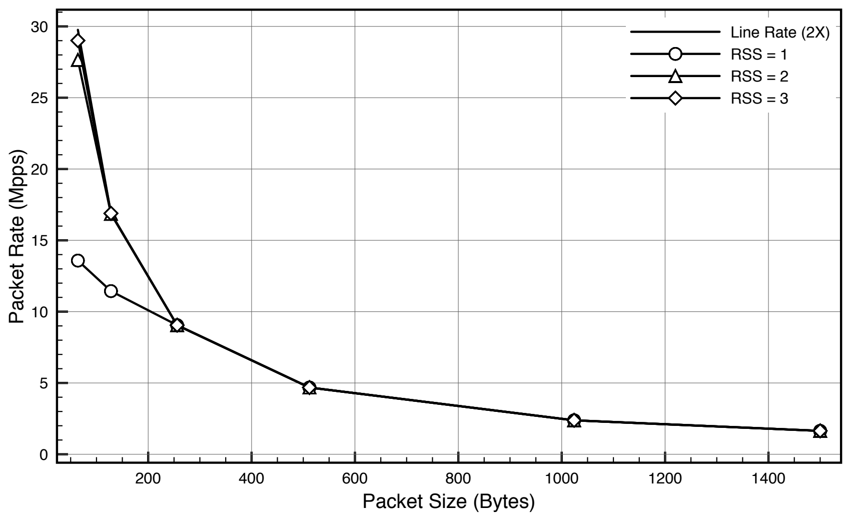


Figure 2.10: *10G packet capture: 2 user space threads (broadcast)*

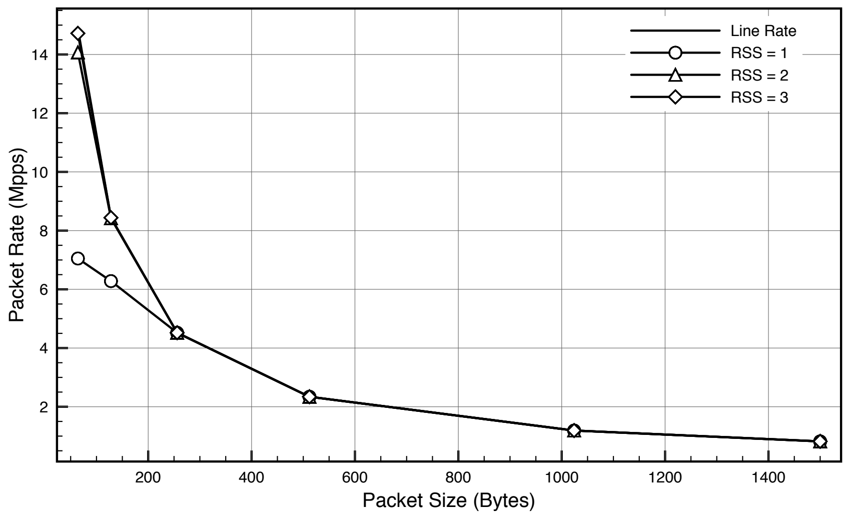


Figure 2.11: *10G packet capture: 2 user space threads (steering)*

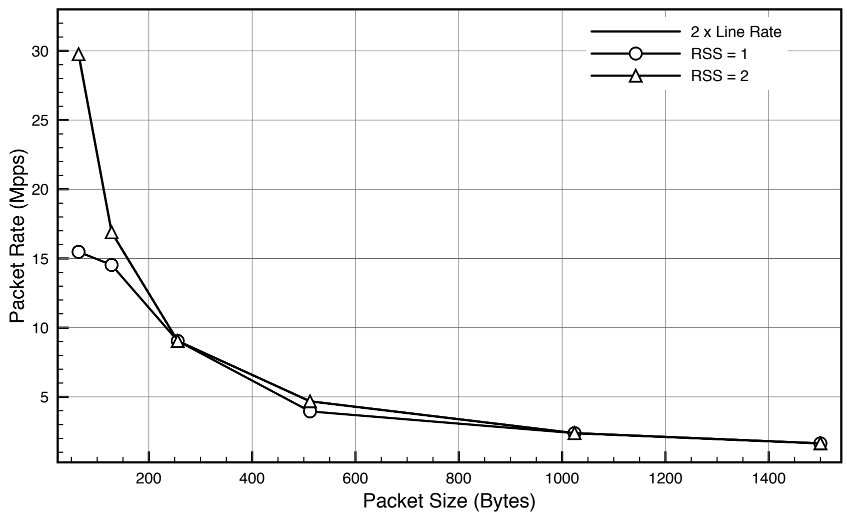


Figure 2.12: *20G packet capture*

When the application threads (belonging to the same monitoring group) become two, a slightly different number of kernel threads is needed due to the upstream delivery mode. Indeed, the broadcast mode (Figure 2.10)) requires PFQ to send a replica of all packets to both threads (which makes an internal throughput of 20 Gbps at full speed), while the steering mode (Figure 2.11)) spreads statistically packets to the application threads. As a result, in our system, broadcasting and steering packets require RSS = 3 to achieve full-rate figures for all packet sizes. It is worth noticing that in case of multiple application threads, a small overhead is also introduced by the functional computations in charge of distributing packets. This is the reason why the extra core is necessary concerning the previous results of Figure 2.9.

2.7.3 Up to 40G Speed Tests

PFQ capturing performance has also been checked for traffic rates of 20 and 40 Gbps.

The 20 Gbps performance test has been carried out by making `pfq-counters` use a single thread to capture traffic from two 10G network interfaces at the same time. In case of tapping traffic from multiple devices, particular attention must be paid in the configuration of interrupt affinities (set with the handful tool irq-affinity, shipped with the framework).

Although it would be possible to use the same kernel threads to fetch packets from both NICs (suffering a sluggish performance), Figure 2.12 reports the performance with RSS set to 2 and each MSI-X interrupts bound to different cores (which makes a total of 4 cores in use).

The results are consistent with the ones shown in Figure 2.9 and demonstrate that PFQ can seamlessly handle two 10G interfaces independently and reach line rate in capturing traffic from each of them by using two cores.

When link speed increases up to 40 Gbps, capture performance does not scale as well. Figure 2.13 shows transmission and capture performance of PFQ by using both the 1.2.48 and the recently released 1.3.47 versions of the Intel i40e driver. Although

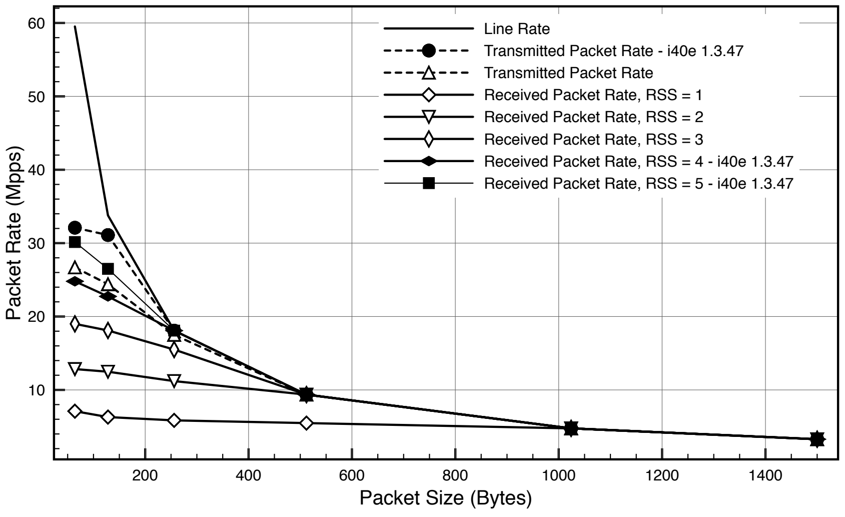


Figure 2.13: *40G packet capture*

Intel claims that XL710QDA2 40G NICs can reach full speed with 128 bytes packet size, our results show that only PFQ transmission performance gets close to line rate with such a packet length (with the new driver), while full rate capture is reached for packet sizes bigger or equal to 256 bytes. By carefully looking at the figure, another interesting point comes out as well: the sustained packet rate achieved with two hardware queues (RSS=2) is lower than that of 10 Gbps interfaces! However, PFQ is driver agnostic and does not introduce modifications in its internal mechanisms when the underlying network devices change. The different behavior of 1.2.48 and 1.3.47 driver versions, however, offers a possible interpretation of such an "inconsistent" behavior. Indeed, up to RSS = 3, the two drivers perform similarly, while version 1.3.47 improves the sustained rate by increasing the number of capturing cores up to 5. Version 1.2.48, instead, does not show any performance improvement for RSS bigger than 3 (the corresponding plots are therefore omitted). This evidence, far from being a definitive proof, suggests that the observed low capture figures (at small packet sizes) may be caused by limitations still present in the i40e driver. We expect that performance will significantly improve as the driver will reach a maturity level compared to that of the ixgbe for the 10 Gbps cards. Conversely, the above results confirm that one of the leading "pros" of PFQ, namely its full hardware transparency, may turn into a "cons" as its performance can be significantly affected by the underlying driver efficiency.

2.7.4 Software Acceleration

The performance results so far presented have been obtained with PFQ parameter configuration finely tuned. Such parameters are strictly connected to the software acceleration mechanisms presented in section 2.3 and in section 2.6. The following experiments aim at evaluating the impact of such acceleration techniques on the overall PFQ performance.

Figure 2.14 shows the effectiveness of using the pool of `skbuffs` in boosting capturing performance. Interestingly, the achieved throughput increases up to a pool size

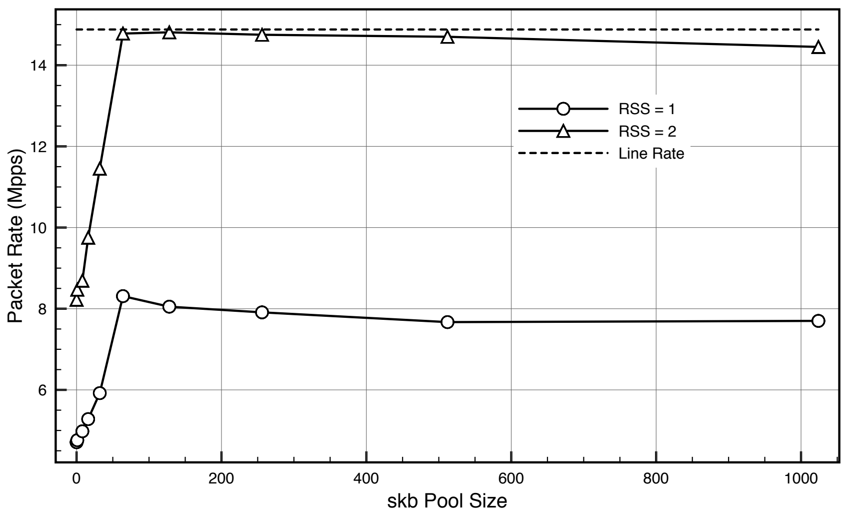


Figure 2.14: *Skb pool acceleration*

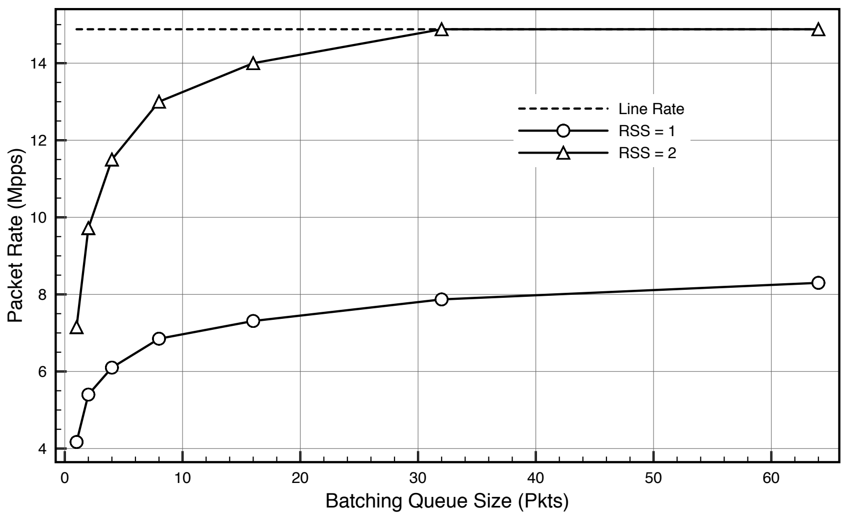


Figure 2.15: *Batch queue acceleration*

of 64 `skbuffs` and then slightly decreases. The most plausible reason for such a slight performance drop may be found in the way packet payloads are stored upon reception. Indeed, the ixgbe driver allocates the DMA addressable memory in pages of 4096 bytes that can accommodate up to 64 packets of 64-byte size. When the number of `skbuffs` exceeds such amount, additional pages must be used, and this may lead to a little performance decrease.

Similarly, Figure 2.15 depicts the beneficial effect of packet batching in the cases of one or two cores devoted to capturing. The results show that performance increases by enlarging the queue length and that a batch size value of 32 packets is enough to reach the maximum benefit (line rate speed in case of RSS=2).

The beneficial effect of batch transmission is, instead, reported in Figure 2.16. Once

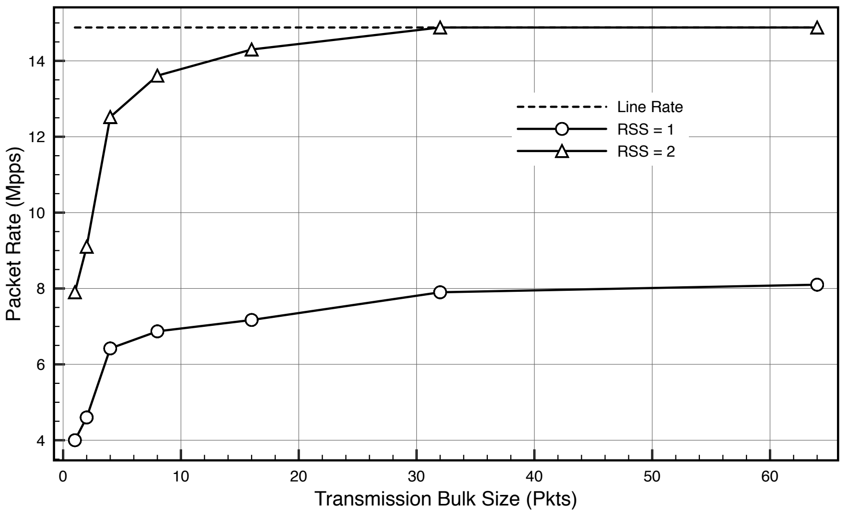


Figure 2.16: *Bulk network packet transmission*

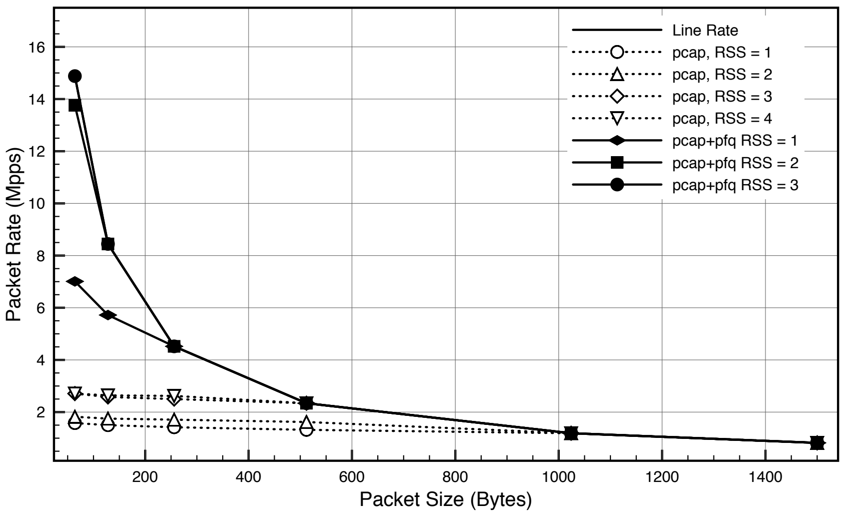


Figure 2.17: *Libpcap acceleration*

again, performance improves by increasing the transmission bulk size until it reaches a plateau at the value of 32 at which driver resources clearly saturate.

2.7.5 Libpcap acceleration

The last performance experiment of this section aims at evaluating the effectiveness of PFQ in accelerating the pcap library. A direct comparison against classic pcap library is possible in that, similarly to PFQ, the underlying Linux PF_PACKET socket can take advantage of the multi-queue support provided by the RSS technology. Hence, the kernel-level capture system can be set to the parallel scheme of Figure 2.3(c) if the interrupt affinity is properly configured.

Figure 2.17 shows the results achieved by using a small pcap application that simply

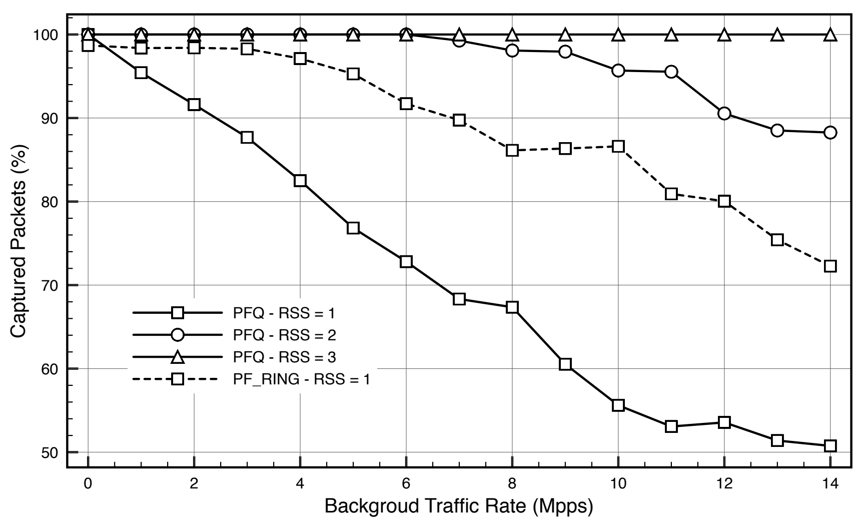


Figure 2.18: *BPF filtered IP traffic with* `tcpdump`

counts packets when running on top of the standard PF_PACKET socket and top of PFQ, under different traffic packet sizes and different RSS values.

The performance improvement is evident and shows that PFQ can effectively accelerate legacy network applications traditionally based on the pcap library. Also, it is worth noticing that the multi-queue support provided by the Intel NICs does not significantly improve the capturing performance of PF_PACKET socket that, indeed, hardly hits 2 Mpps rate, even with four hardware queues (RSS = 4).

The most likely reason for the observed libpcap performance resides in the implementation of the PF_PACKET socket and involves explicitly the re-aggregation in a single queue of skbuffs coming from different queues/cores. Indeed, this operation is more efficient in PFQ because of its total lock-free architecture and because of the batch-fashion policy adopted to amortize atomic operations. Notably, both PFQ and PF_PACKET use a memory mapped area shared between kernel-space and user-space where packet payloads are copied. However, PFQ uses HugePages (if correctly configured), whereas PF_PACKET uses standard 4k system pages.

2.8 Use-cases

This section presents the use of PFQ in four practical use-cases involving real network applications with increasing complexity. Along with showing mere performance results, the other – and somewhat primary – objective of the following presentation is to evidence the high usability and flexibility of PFQ in common monitoring scenarios where fine-grained parallel processing schemes are often necessary. The first three use-cases are related to new and legacy network applications monitoring traffic over a 10 Gbps link. The last use-case, instead, deals with packet transmission and reports on the acceleration of the widely known traffic generator *Ostinato* [83].

2.8.1 IP address traffic filtering

In this use case, a single instance of the well known `tcpdump` sniffer is used to tap real packets of variable length and source IP address set to 1.1.1.1 out of a synthetic aggregate of 64 bytes long UDP packets. The real trace is played at 1 Gbps while background traffic is played at increasing packet rates, up to link saturation. `tcpdump` runs on top of the pcap adaptation layer of PFQ and packets are filtered using native BPFs. All this is instantiated through the following simple command line:

`tcpdump -n -i pfq:eth3 "host 1.1.1.1"` by which the application registers to the first free group available in the system on the network interface eth3 and specifies the BPF filter “host 1.1.1.1”.

Figure 2.18 shows (in solid lines) the number of packets received by the sniffer under a different number of enabled hardware queues. Notice that RSS = 2 is sufficient to reach full rate filtering up to 6 Mpps of disturbing traffic but the small overhead introduced by both the pcap adaptation layer and the BPF filter requires an extra core to achieve full capture rate in all conditions.

Further, Figure 2.18 also reports (in broken lines) the performance of PF_RING ZC in filtering IP packets. This plot is reported to make clearer the trade-offs introduced by the PFQ architecture concerning a well-known alternative high-performing capture engine. The figure shows that PF_RING ZC can sustain a higher traffic rate with one single core (RSS = 1), reaching more than 70% capture rate in the worst case. However, to achieve 100% rate, PF_RING would need the use of two cores which, in turn, would require either to run two instances of `tcpdump` or to implement an ad hoc software module to re-aggregate at the user-space the packets previously spread out by the RSS algorithm. PFQ (and its pcap adaptation layer), instead, dispatches and re-aggregates packets transparently to user applications: this allows a single instance of `tcpdump` to receive all traffic at the cost of an extra core (RSS = 3) to reach full rate performance.

2.8.2 RTP flow analysis

In the second use case, the pcap based `tshark` sniffer [107] (from the *wireshark* package) is used to capture and reconstruct RTP audio flows played at different speeds on a 10G link up to link saturation, with or without the underlying PFQ support. To this purpose, an instance of `tshark` runs with the following command line:

```
tshark -i eth3 -z rtp,streams -q
```

Figure 2.19 shows that `tshark` alone does not catch up with the input traffic pace and the percentage of captured packets rapidly drops below 50% as the input rate increases.

When PFQ is enabled in the optimal setup of RSS = 2 (as experimentally determined for this test), `tshark` performance quickly increases, although a small percentage of packets are dropped at high traffic speeds. Such packets are not dropped by PFQ (which can easily handle the 10G rate of packets longer than 64 bytes). In fact, packets are dropped at the user-space by `tshark` itself that cannot accomplish its computations at such a high packet rate. Using PFQ, this problem can be elegantly overcome by increasing the number of userspace instances of `tshark` and by letting them join the

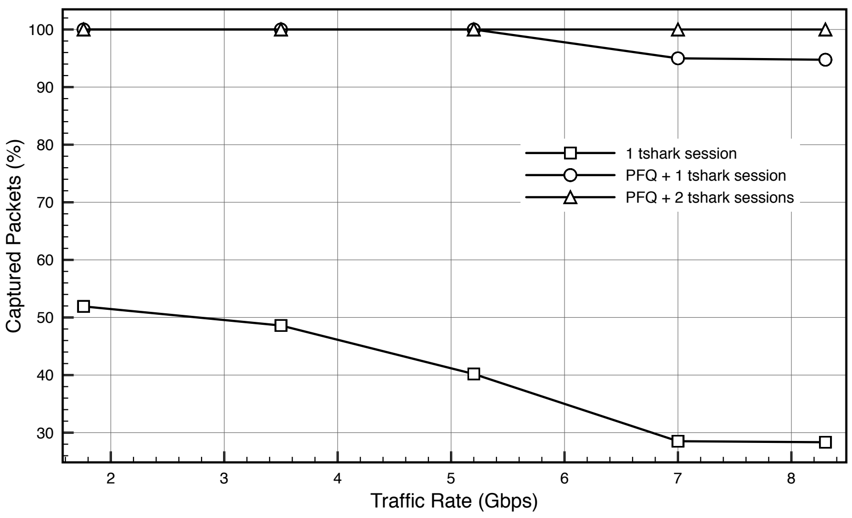


Figure 2.19: *RTP flow processing with* `tshark`

same group. As a result, the top graph of figure 2.19 shows that the two instances achieve 100% capture rate by receiving around half of the overall packets to process.

The operation is easily accomplished by launching each of the `tshark` instances through the following command line:

```
LD_PRELOAD=/usr/local/lib/libpcap-pfq.so
     PFQ_CONFIG=/etc/pfq.pcap
     tshark -i eth3 -z rtp,streams -q
```

that instructs `tshark` to use the pcap adaptation layer of PFQ and to retrieve group parameters and the `steer_rtp` computation that broadcasts RTCP packets to a specific control-plane class, and dispatch RTP flows to the user-plane one.

Although the above procedure allows spreading RTP/RTCP traffic to multiple process instances, it still does not permit `tshark` to properly reconstruct flows and prepare a statistic summary. In fact, to accomplish such operations, both `tshark` instances need to access to SIP messages associated with the RTP flows. PFQ helps to get around this issues through the use of a more complex computation:

```
steer_voip = conditional' is_sip
    (class' class_control_plane >-> broadcast)
    steer_rtp
```

that allows the two instances of `tshark` to receive (in broadcast) a copy of all SIP packets as a result of the convenient SIP filtering computation.

The results, shown in Figure 2.19, are obtained under this setup with the two instances of `tshark` reporting the RTP flow summary statistics at the end of their elaborations on the received traffic.

2.8.3 LTE analyzer

The last monitoring use-case consists of a multi-threaded application designed to natively run on top of PFQ to perform per-user LTE traffic analysis and provide statistics, such as number of packets sent and received, per-user TCP flow count, TCP packet retransmission, etc.. In addition, the application runs some basic security algorithms (e.g., SYN flood detection) and user protocol classification (through OpenDPI).

As a result, concerning the previous scenarios, this application represents a significant step ahead in benchmarking PFQ features and performance, regarding both (higher) computation resources and fine-grained functional requirements.

Indeed, to complete the overall amount of computations, LTE analyzer needs, on average, a few (around 10) thousands of clock cycles per each processed packet. As it will be shown later on, this requires multiple threads to catch up with high traffic rates. Regarding functional features, instead, the application requirements are directly induced by the way LTE user plane (UP) traffic is carried over IP.

LTE packets are transported over GTP v1/2 tunnels. As such, per-user analyses cannot leverage on the rough parallel schemes provided by RSS, nor on steering functions that spread traffic to application threads by hashing over the canonical IP 5-tuple. The functional computations of PFQ must, in this case, delve into the payload of GTP packets and access user-information to distribute packets to upstream applications. Besides, all application threads must access the GTP control plane data (CP) to accomplish their analysis.

The above requirements are readily met through the GTP related computations at the kernel level, and using groups, classes and their configurable delivery mode for packet distribution.

The results, shown in Figure 2.20, are obtained under a different number of applications threads (sharing a common monitoring group and registered to a common control plane class) that receive:

- UP packets on a per-user basis through the `gtp_steer` kernel computation, in steering mode;
- all of the CP packets, in broadcasting mode.

A real GTP trace is played by `pfq-gen` at different speeds over a 10 Gbps link up to its saturation. PFQ is optimally configured to use two functional engines (RSS=2) that do not overlap those running the application threads. Figure 2.20 shows the percentage of received UP packets retrieved by the LTE analyzer application concerning the input traffic rate and for a different number of application threads.

The significant computation machinery of the application does not allow a single thread to sustain more than 4 Gbps input traffic. This is a classic case in which the only way to scale performance is to take advantage of parallelism. In our case, it takes up to 3 application threads (though two threads slightly suffer at full line rate only) to sustain a traffic rate of 10 Gbps.

2.8.4 Accelerated Traffic Generation

The last use-case refers to packet transmission and aims at assessing the effectiveness of PFQ in accelerating the well-known traffic generator `Ostinato`. Ostinato is a highly configurable open source traffic generator that supports a wide variety of protocol templates for packet crafting. It is based on a client-server architecture; the server (drone) runs each engine as a single-thread of execution that uses the pcap library for packet transmission.

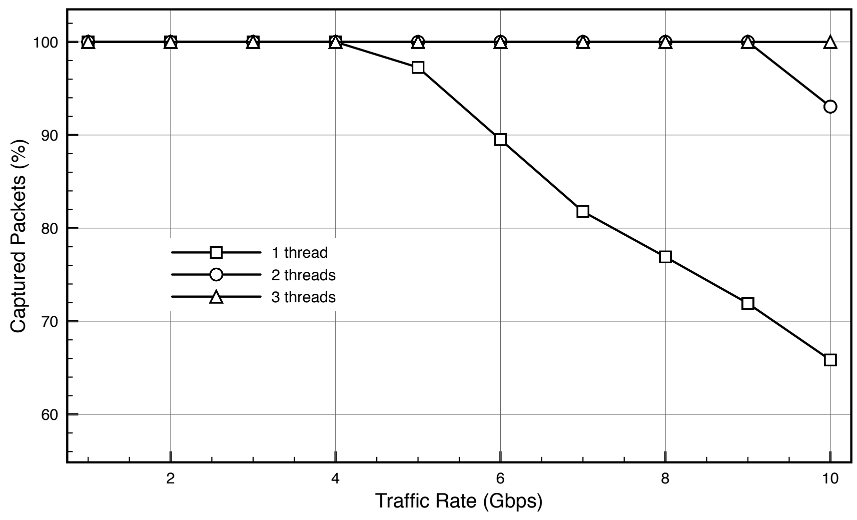


Figure 2.20: *LTE analyzer*

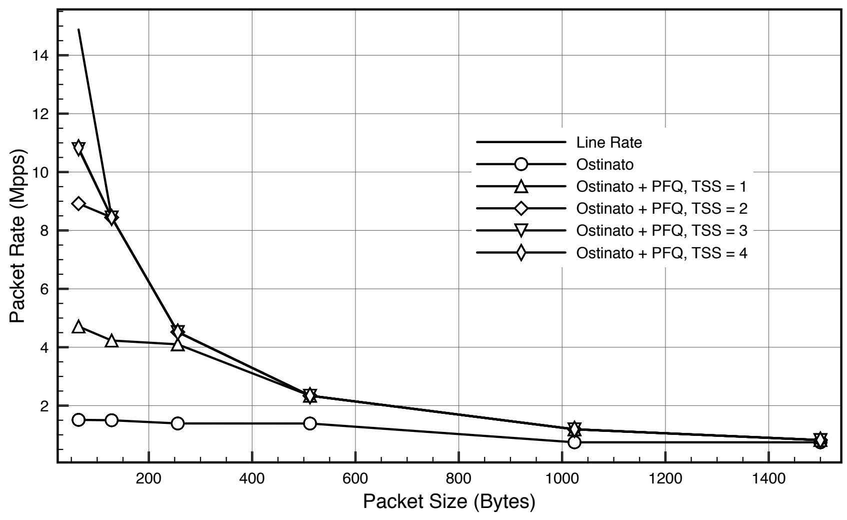


Figure 2.21: *Ostinato packet transmission acceleration with PFQ*

Figure 2.21 shows the result of the experiment. Ostinato was first executed alone with the optimal value of 4 hardware queues for transmission (although, as shown in section 2.7.5, the number of hardware queues used by the standard PF_PACKET socket does not make significant differences). The results show that Ostinato alone can hardly reach near full rate generation speed in the only case of 1500 bytes long packets. In all other cases, its performance is far from the theoretical physical limit.

The use of PFQ significantly accelerates the application performance, although line rate is achieved for packet sizes of at least 128 bytes. However, even in the worst case of 64 bytes long packets, PFQ allows bringing the Ostinato performance above 10 Mpps transmission rate (i.e., yielding an acceleration factor slightly larger than 7) with three transmitting kernel threads and affinity setup that preserves the engines from running the Ostinato drone itself. Conversely, the figure also shows that no significant improvement can be noticed by increasing the number of transmitting cores beyond 4.

CHAPTER 3

Traffic Distribution

3.1 Introduction and motivation

The technological maturity reached in the last years by general purpose hardware is pushing commodity PCs as viable platforms for running a whole bunch of network applications devoted to traffic monitoring and processing. Indeed, the availability of 10+ multi-gigabit network cards allows to easily connect a standard PC to high-speed communication links and potentially retrieve huge volumes [34] of heterogeneous traffic streams.

In the last few years, the computational power provided by the always increasing number of cores available on affordable CPUs combined with the hardware multi-queue support of modern network cards has favored a large interest in the research community towards software accelerated solutions for efficient traffic handling on traditional PCs running Unix Operating Systems.

As a result, to date, capturing packets at full-rate over multi-gigabit links is no longer an issue and it is made possible by several alternatives *packet I/O frameworks*, each of them with its own set of features. However, the higher packet rate attained by the accelerated capture engines may not, by itself, guarantee better application performance. Indeed, computation intensive operations such as those performed by classical network monitoring applications, Intrusion Detection and Prevention Systems, routers, firewall and so on, do not often catch up even with the non-accelerated traffic rates provided by the standard sockets. In all such cases, the use of accelerated capture engines does not give any benefit as the application would get overwhelmed by an excessive amount of packets that cannot be handled. In fact, in many cases, the overall performance may even further degrade as the extra CPU power consumed to accelerate capture operations is no longer available for the application processing.

When the performance bottleneck is represented by the application itself, the straightforward way of scaling up performance is leveraging on *computational parallelism* by spreading out the total workload over multiple *workers* running on top of different cores. This, in turn, requires on one hand network applications to be designed according to multi-thread/multi-process paradigm and, on the other hand, the underlying capture technology to support *packet fanout* to split and distribute the total workload among multiple workers. Currently, albeit with different features and programmable options, both standard and accelerated sockets support packet fanout. Unfortunately, most of today's network applications are still single-threaded and access live traffic data through the `pcap` library (`libpcap`) [89] rather than using the underlying sockets. Over the years, the `libpcap` library has emerged as the, somewhat, de-facto standard interface for handling traffic data and, as it will be shown, its use has many practical advantages. However, the current `pcap` library does not support packet fanout, thus preventing transparent applications parallelism.

The objective of this work is to present the implementation of a new `pcap` library for the Linux operating system that supports packet fanout while still retaining full backward compatibility with the current version.

The new library is freely available for download[1]

The chapter extends the previous conference version [22] in several different directions. First, the new `pcap` library itself has been extended with a set of APIs to simplify its use in practical multicore scenarios. The applicability of the library has also been broadened to include the explicit support of a full set of accelerated sockets. Besides, the ongoing research on a unified solution for nearly-agnostic support of any underlying sockets is also given. The experimental part has been significantly extended by including new sockets in the set of performance tests as well as adding more realist traffic scenarios in the library assessment in practical use-cases.

More in detail, the standard scheme for accessing live network data on Linux is first presented in section 3.2. This includes a description of the standard Linux socket and its packet fanout features as well as a brief introduction to the use of the `pcap` library. Section 3.3 presents a concise overview of the available accelerated capture engines together with their classification into the two broad categories of *active* and *passive* sockets. The section discusses explicitly the different issues that emerge when trying to enable packet fanout in both classes and provides the reasons for the current support of active sockets only. Two specific engines (PF_RING and PFQ) from this category are then briefly introduced as they will be next used in the experiments to improve the performance of applications using the newly developed library. Section 3.4 represents the core of the chapter and includes the description of the library for parallelizing native and legacy applications in both standard and accelerated scenarios. Section 3.5 presents a discussion on how to practically configure software and hardware resources to effectively take advantage of the new features available from the library. Experimental assessment is carried out in sections 3.6 and 3.7 that reports on the performance improvement brought by the new library in pure speed tests and practical use cases involving the well-known applications `Tstat` and `Bro`, respectively. Section 3.8 elaborates upon the extensions needed to provide the `pcap` library with the support for passive sockets and presents the design of a possible unifying architecture whose development is currently ongoing.

[1]repository at https://github.com/awgn/libpcap, branch 'fanout'

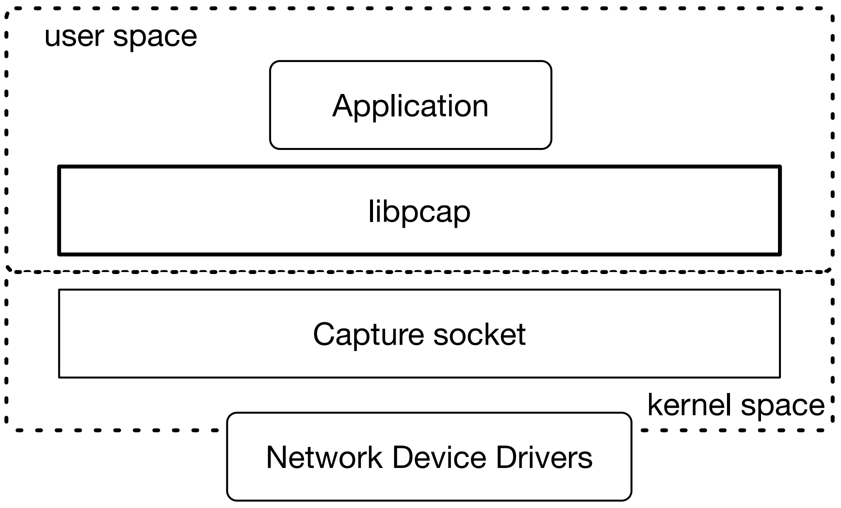


Figure 3.1: *Network application stack*

3.2 Packet Dispatching in Linux

The typical scheme of a network application handling live traffic in the Linux operating system is shown in Figure 3.1. Upon their arrival at the physical interface, packets are managed by the device drivers and made available to the application through *packet sockets*. The low-level handling of the socket operations can either be performed by the application through the native socket API or be left to the `pcap` library interface. This section aims at describing the main internals of the default Linux socket with a specific focus on the less known packet dispatching features. The use of the `pcap` library is also briefly introduced to point out its current limitations that motivate this work to achieve a full integration with the standard socket features.

3.2.1 Linux Default Capture Socket

The default Linux socket for packet capture is the `AF_PACKET` socket and its more efficient memory mapped variant `TPACKET` (currently at version 3).

At the lower level, both `TPACKET` and `AF_PACKET` support multi-core packet capturing, that is they take advantage of Received Side Scaling technology (RSS) [59] to retrieve packets in parallel from multiple hardware queues of network interfaces as shown in Figure 3.2.

Since kernel version 3.1, to scale processing across up-layer computing workers, the standard Linux socket supports configurable packet fanout to multiple sockets through the abstraction of *fanout group*. Each thread/process in charge of processing traffic from a network device opens a packet socket and *joins* a common fanout group: as a result, each matching packet is queued onto only one socket in the group, and the total workload is spread upon the total number of instantiated threads/processes.

Groups are implicitly created by the first packet socket joining a group, and the maximum number of groups per network device is 65536. Sockets join a fanout group by means of the `setsockopt` system call with the `PACKET_fanout` option. Conversely, packet sockets can leave a group only by closing the socket. When the last

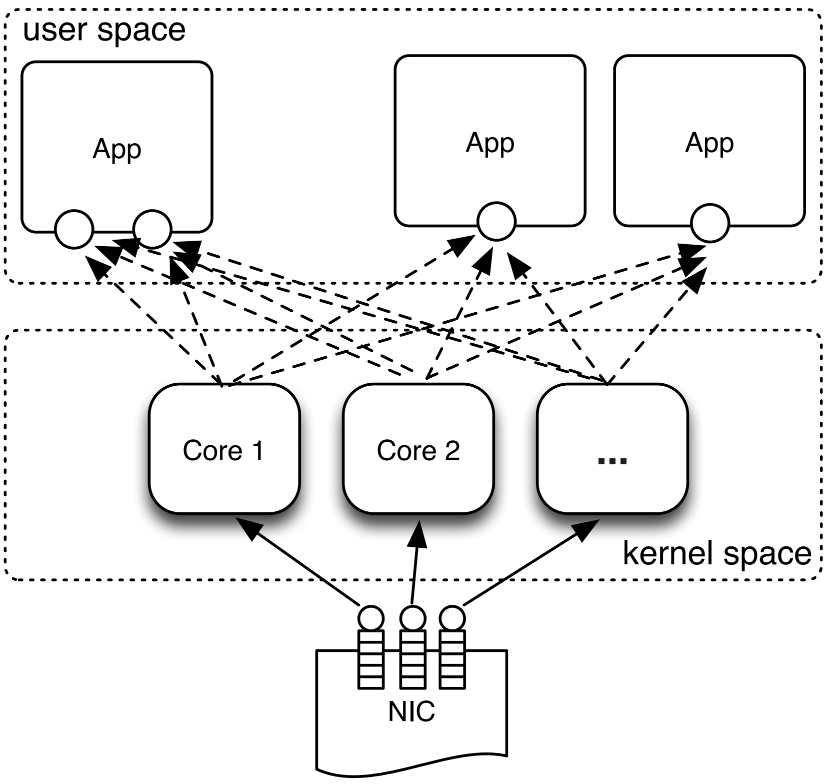


Figure 3.2: *Standard Linux socket*

socket registered to a group is closed, the group is deleted as well. Finally, to join an existing group, the next packet sockets must obey the set of standard settings already specified for the group, including the *fanout mode*.

3.2.2 Socket Fanout Modes

Packet fanout is the straightforward solution to scale processing performance by distributing traffic workload across multiple threads/processes. The criteria in which packets are spread out among the workers have a significant impact in both functional and performance points of view.

The standard Linux socket supports a limited number of algorithms (*modes*) for traffic distribution. The available fanout modes are presented in the following list.

- The default mode, namely `PACKET_FANOUT_HASH`, preserves flow consistency by sending packets from the same flow to the same packet socket. Practically, a hash function is computed over the network layer address and (optionally) transport layer port fields. The result (modulo the number of sockets participating the group) is used to select which socket to send the packet to.

- The `PACKET_FANOUT_LB` mode simply implements a round-robin load-balancing scheme to choose the destination socket. This mode is suited for purely stateless processing as no flow consistency is preserved.

- The PACKET_FANOUT_RND mode selects the destination socket by using a pseudo-random number generator. Again, this mode only allows stateless processing.
- The PACKET_FANOUT_CPU mode selects the packet socket based on the CPU that received the packet.
- The PACKET_FANOUT_ROLLOVER mode keeps sending all data to a single socket until it becomes backlogged. Then, it moves forward to the next socket in the group until its exhaustion, and so on.
- The PACKET_FANOUT_QM mode selects the packet socket whose number matches the hardware queue where the packet has been received.

3.2.3 Standard pcap interface

Most of the more popular network monitoring applications (such as *tcpdump*, *wireshark*, etc.) are written on top of the `pcap` library [89]. As depicted in Figure 3.1, the `libpcap` layer hides low-level traffic capture details to the upper layer application by providing a standard and unified API for generic packet retrieval and handling. As such, the use of `libpcap` eases application portability and adds useful features such as read/write access to trace files and packet filtering using *Berkeley Packet Filters* (BPF).

However, as a major drawback, the `pcap` library lacks the native support for multi-thread programming. This reason forces developers that need to implement schemes such as the one shown in Figure 3.3 to provide an additional layer of packet distribution built into the applications. By default, both threads of the Application 1 would receive a replica of the same traffic, and so would the two instances of the Application 2. This design looks even more paradoxical as the default socket used by `libpcap` in the Linux version (`TPACKET`) supports indeed packet fanout. As will be elaborated, the primary objective of this work is to remove this limitation by providing the fanout support to the `libpcap` interface.

3.3 Software acceleration

In the previous sections, packet fanout has been introduced as the straightforward way of scaling performance by splitting traffic workload among multiple workers, typically running on different cores or CPUs. However, when link speeds raise up to multi-gigabit rates, the default sockets may not be able to catch up with the actual packet arrival rate, causing a significant drop rate at the physical interfaces. In all such cases, the use of *accelerated capture sockets* is mandatory to increase the number of packets captured on the wire and dispatched to the application workers. Notice, however, that packet capture and packet distribution to up-layer software are independent operations and very efficient capture sockets may not necessarily support fanout algorithms.

In the last few years, a significant number of accelerated sockets have been proposed for efficient traffic capture at 10G+ links speed (see references [26,51,77] for a thorough overview). One of the first software accelerated engines was PF_RING [50] which proved to be quite successful in case of 1 Gbps links. PF_RING uses a memory mapped ring to export packets to user-space processes and supports both vanilla ("classic") and modified ("aware") drivers. More recently, PF_RING ZC (Zero Copy) [39], and

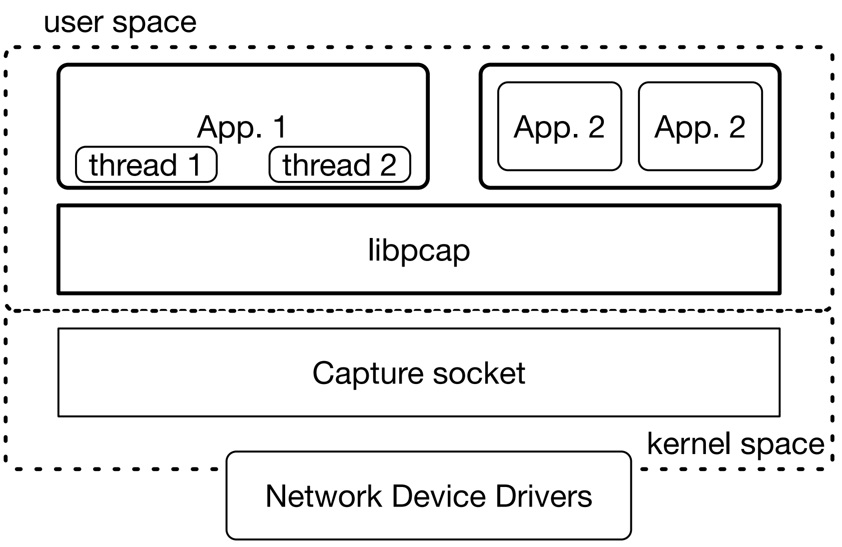


Figure 3.3: *Multi-workers network applications*

Netmap [93], allow a single CPU to retrieve short sized packets up to full 10 Gbps line rate by memory mapping the ring descriptors of NICs at the user space. DPDK [42] is another successful solution that bypasses the operating system to accelerate packet capture. DPDK provides a Linux user-space framework for efficient packet processing on multi-core architectures based on pipeline schemes. Finally, PFQ [19] is a software acceleration engine built upon standard network device drivers that primarily focuses on programmable packet fanout.

Generally speaking, accelerated sockets can be divided into two broad categories to distinguish those that use an active context to fetch packets from the network card from those that, instead, execute network device drivers in the calling context – usually the user-space process. We name the sockets of the first category as *active sockets* since they revolve around the concept of a running context (e.g., the NAPI kernel thread). Conversely, we name as *passive* the sockets that fall into the second category.

Among the above listed accelerated sockets, the active sockets category includes the standard Linux PF_PACKET/TPACKET3, PFQ, and PF_RING (both "classic" and "aware" flavor). Instead, PF_RING DNA, PF_RING ZC, Netmap, and DPDK belong to the class of passive sockets.

Although under different names, all of the active sockets support packet fanout in kernel space within the NAPI soft IRQ context. At this stage, the different implementations allow distributing the incoming packets to a group of sockets, by applying different balancing schemes.

Passive sockets, instead, target top performance by removing the IRQ latency and thus relying on a more aggressive polling which executes in the caller context directly[2]. This approach prevents implementing packet distribution algorithms except for very few commercial applications (such as the `libzero` library of PF_RING ZC) that offer similar support.

[2]Aggressive polling is required to cope with the limited amount of packet descriptors available in commodity NICs.

In other words, parallelizing a network application on top of a passive socket over multiple working threads/processes requires the application itself to implement a suitable packet distribution scheme. This, in turn, requires a significant rewrite of the application code to implement a receiving thread that polls the NIC and performs packet steering, lock-free queues for packets passing, and so forth – for example, these are typical issues to be handled when using DPDK. However, this approach harshly clashes with the design philosophy of both the original `pcap` library, which aims at simplifying the life of applications in capturing/injecting packets, and our variant version that, in addition, target performance scaling by means of the fanout feature with no modifications to the application source code.

For all the above reasons, the current version of the new `pcap` library implements the fanout feature for active sockets only, i.e., the standard Linux socket, PF_RING and PFQ. To include the support of passive sockets within the same semantic, an additional *fanout abstraction layer (FAL)* is required. Although not yet fully implemented, a brief description of the preliminary architecture of the FAL is reported in section 3.8.

3.3.1 The PF_RING accelerated socket

PF_RING is a popular family of accelerated sockets. The family includes a variety of sockets (PF_RING, PF_RING DNA, PF_RING ZC), each with different network device drivers and internal semantics. All the PF_RING variants are supported by a custom `pcap` library implemented by the maintainers that makes it easily pluggable into legacy applications.

As shown in Figure 3.4, the *classic* PF_RING (also known as *vanilla* PF_RING) is an active socket that polls packet from the NIC through the Linux NAPI. Packets are then copied into circular buffers that are memory mapped to the user-space for application consumption. As such, PF_RING allows workload distribution to multiple rings (hence, multiple applications) and support packet fanout through the concept of *clustering*. PF_RING clusters are very similar to TPACKET and PFQ groups. Indeed, a set of applications sharing the same *clusterId* receive packets coming from one or more ingress interfaces according to the different balancing algorithms reported in Table 3.1.

In the recent past, the PF_RING package contained a set of hardware-specific optimized (*aware*) device drivers for several NICs that significantly increased capturing performance. To date, classic PF_RING ships with vanilla driver only, while very top performance is left to the PF_RING ZC passive socket.

3.3.2 The PFQ accelerated socket

The architecture of PFQ as a whole has been already described in chapter 2 and is sketched in figure 2.1.

It's worth mentioning here that the main objective of PFQ is to capture and retrieve packets from *one or more* traffic *sources*, makes some *computations* with functional blocks (the λ_i blocks in the picture) and deliver them to *endpoints*.

Similarly to the Linux socket, PFQ uses the abstraction of groups as a set of sockets that share a computation and the same input sources. From the application point of view, user-space threads open such sockets to *join* a group and capture packets steered from kernel–space by PFQ-Lang computations.

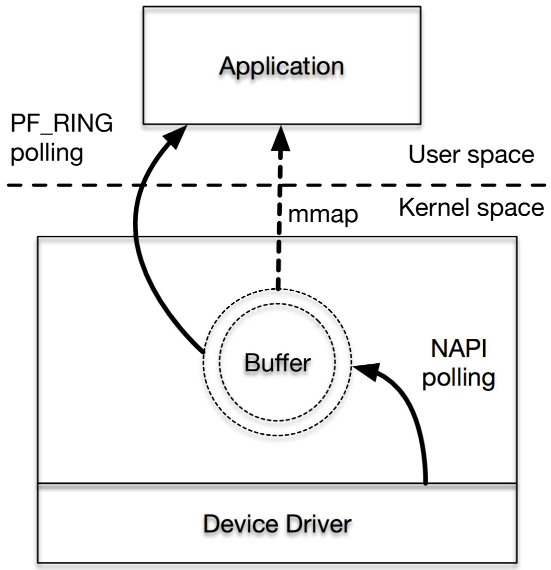


Figure 3.4: *The PF_RING socket*

Table 3.1: *Packet fanout modes in PF_RING*

PF_RING balancing mode	Description
`round_robin`	Sends packets to sockets according to round robin algorithm
`flow`	Sends packets to sockets according to the hash value computed over the 6-tuple <src ip, src port, dst ip, dst port, proto, vlan>
`flow_2_tuple`	Sends packets to sockets according to the hash value computed over source and destination IP addresses
`flow_4_tuple`	Sends packets to sockets according to the hash value computed over source/destination IP addresses and source/destination ports
`flow_5_tuple`	Sends packets to sockets according to the hash value computed over source/destination IP addresses, source/destination ports and the protocol field
`flow_tcp_5_tuple`	Same as 5_tuple for tcp protocol, 2_tuple otherwise
`inner_flow`	Same as `flow`, only for packet headers transported by tunnel
`inner_flow_2_tuple`	Same as `flow_2_tuple`, only for packet headers transported by tunnel
`inner_flow_4_tuple`	Same as `flow_4_tuple`, only for packet headers transported by tunnel
`inner_flow_5_tuple`	Same as `flow_5_tuple`, only for packet headers transported by tunnel
`inner_flow_tcp_5_tuple`	Same as `flow_tcp_5_tuple`, only for packet headers transported by tunnel

The steering functions of PFQ are in some way related to the concept of fanout, as they fulfill the principle of distributing packets, ensuring in addition different degrees of flow consistency.

Table 3.2: *Packet fanout modes in PFQ*

PFQ steering function	Description
`steer_rrobin`	Sends packets to sockets according to round robin algorithm
`steer_rss`	Sends packets to sockets according to the RSS hash value computed by the device driver
`steer_rx_queue`	Sends packets to the sockets with index matching the hardware queue index
`steer_link`	Send packets to the sockets preserving coherency at link-layer
`steer_local_link`	Like above but with support of double-steering
`steer_vlan`	Sends packets according to vlan tag value
`steer_p2p`	Sends packets according to the symmetric hash value computed on the pair of source/destination IP addresses
`steer_local_ip`	Like above but with support of double-steering for local traffic
`steer_flow`	Sends packets according to the hash value computed on the packet flow headers
`steer_to`	Sends packets deterministically to a specific socket
`steer_field`	Sends packets according to the hash value computed on the specified field
`steer_field_symmetric`	Sends packets according to the symmetric hash value computed on a pair of specified fields
`double_steer_mac`	Sends packets according to the symmetric hash value computed on the pair source/destination mac addresses
`double_steer_ip`	Network internal packet (local IP to local IP) are doubly dispatched on the basis of the pair source/destination Ip addresses
`double_steer_field`	Packet are doubly dispatched on the basis of the specified pair of fields

Table 3.2 herein reported lists the main fanout modes already implemented in PFQ and reports, for comparison, (if any) the analogous modes implemented by the standard Linux socket.

3.4 Packet fanout support in the `pcap` interface

As previously mentioned, the current implementation of the `libpcap` library does not provide a dedicated API to facilitate multi-core parallel processing. Therefore the whole traffic stream captured over a physical interface is not split across multiple threads/processes. In fact, multiple workers bound to the same network interface would all receive a replica of the whole amount of traffic captured at the physical device. This section reports on the extension of the existing `pcap` library to enable packet fanout and to provide flexible support for multi-core processing.

The starting point was to comply with the basic operation of the underlying Linux socket `TPACKET` by integrating the notions of *group of sockets* and *fanout modes* into the `pcap` library. This implied a significant reworking throughout the whole library code. However, all the changes are buried into the library implementation, and packet fanout can be enabled through the following *single* API:

```
int pcap_fanout(pcap_t *p,
                int group,
                const char *fanout);
```

Along with the obvious `pcap` descriptor `p`, the function requires specifying the (integer) group identifier and a string representing the fanout mode. The function returns 0 in case of success and -1 in case the operation cannot be completed[3].

The use of this function enables multiple threads of an application to register to a specific group and obtain a quota of the overall traffic according to the selected fanout mode.

When the extended library is used over the standard Linux socket, the fanout mode should be selected among the ones listed in section 3.2.2 and provided by the socket itself. When using an alternative socket, fanout modes must comply with the ones supported by the underlying engine (see Table 3.2 for the main fanout modes available with PFQ and the equivalent supported by `AF_PACKET/TPACKET`).

3.4.1 Legacy application: pcap configuration file

The use of the extended API is well suited when writing a new application in a multithreaded fashion. However, most widely popular network applications are single threaded and their rewriting according to a multithreading paradigm is not feasible in most practical cases.

In all such cases, the extended `pcap` library still allows attaining parallelism by running multiple instances of the same application. All processes that join the same group will then receive a fraction of the total traffic workload, according to a declarative grammar specified in a *configuration file* and without requiring modifications to the application itself.

The grammar of the `pcap` configuration file has the following syntax:

```
key[@group] = value[,value, value...]
```

where the most commonly used keys are:

- `def_group`: default group associated with the configuration file
- `fanout`: string that specifies the fanout mode (example: `fanout = hash`)
- `caplen`: integer values that specifies the capture snaplen (if not specified by the application itself)
- `group_eth<N> = i`: force all sockets bound to the `eth<N>` interface to join group i (example):

  ```
  group_eth0 = 2
  group_eth3 = 3
  ```

[3]The specific error string can still be accessed through the function `pcap_geterr(p)`

Table 3.3: *PCAP environment variables*

Environment Variable	Description
PCAP_DRIVER	Forces the socket type when the device name cannot be mocked (e.g., PCAP_DRIVER=pfq or PCAP_DRIVER=pfring)
PCAP_CONFIG	Overrides the default configuration files which are "/etc/p-cap.conf", "/root/.pcap.conf"
PCAP_GROUP	Specifies the default group for the application... (e.g., PCAP_GROUP=2)
PCAP_GROUP_dev	Specifies the group for the sockets bound to the dev device (e.g., PCAP_GROUP_eth0 = 5)
PCAP_FANOUT	Specifies the fanout algorithm
PCAP_CAPLEN	Overrides the pcap snaplen value
PCAP_CHANNEL_dev	Specifies the number of channels for device <dev> (e.g., RSS)
PCAP_IRQ_dev_0	Sets the IRQ affinity of device <dev> (e.g., eth0) channel 0

Different fanout modes can also be selected for different groups. As an example, the configuration file may contain the following two lines:

```
fanout@2 = hash
fanout@3 = rnd
```

The use of the configuration file is enabled by the environment variable PCAP_CONFIG that contains the full path to the file. The first time it is invoked, the function pcap_activate searches for the presence of the environment variable PCAP_CONFIG. If the variable is specified, the configuration file is open and parsed to retrieve the values of the keys.

Notice that several keys of the configuration file can also be specified on the command line using additional environment variables, with the consequence of overriding the correspondent settings in the configuration file. The set of common environment variables is reported in table 3.3. As an example, a generic instance of the application foo launched as:

```
PCAP_FANOUT="rnd" PCAP_GROUP = 3 foo
```

will receive traffic according to the "rnd" fanout mode on the group 3 regardless of the values specified in the configuration file.

3.4.2 Accelerated configuration

The combined use of environment variables and the configuration file make applications running on top of the new pcap library agnostic to the underlying capture engine and to the way it implements the packet fanout.

As such, although the features of capture engines may be significantly different, the basic semantic of the pcap configuration does not change, and the common set of environment variables reported in table 3.3 can be used irrespective of the underlying technology. However, specific socket-dependent features can still be enabled by relying on the environment variables of the different engines and still left available for compatibility reasons.

This section describes the specific configurations needed to use the pcap library on top of the PF_RING and PFQ sockets. However, it is worth pointing out that similar arguments may be applied to other possible accelerated capture engines if adequately integrated.

PF_RING configuration.

The standard Linux socket can be effortlessly replaced by PF_RING by merely prefixing with the string "pfring" the names of the network devices to be monitored (as an example, `pfring:eth3`). The semantics of the new `pcap` library allows to configure the system by selecting the fanout algorithm within the ones listed in table 3.1 and by choosing the monitoring group of the applications which are transparently mapped into cluster ID of PF_RING. As an example, the following two lines enable two sessions of `tcpdump` to receive a *round robin* share of the packets arriving at the network interface `eth3` on the common group 42.

```
PCAP_FANOUT="round_robin" PCAP_GROUP = 42
            tcpdump -i pfring:eth3
PCAP_FANOUT="round_robin" PCAP_GROUP = 42
            tcpdump -i pfring:eth3
```

Finally, it is worth noticing that even the PF_RING Zero Copy (ZC) passive socket is supported through the same semantic and can be activated by simply prefixing with the string "zc" the name of the network card. However, as discussed in section 3.3, packet fanout is not available in this case as PF_RING ZC does not implement clustering at a lower level.

PFQ configuration.

Similarly to PF_RING, the PFQ socket is enabled whenever the network device name is prefixed by the string "pfq". However, for applications that do not allow arbitrary names for physical devices, it can still be enabled by specifying the name of the driver by setting PCAP_DRIVER=pfq.

The general syntax of the device name is the following:

```
pfq:[device[^device..]]
```

where the character ^ is used to separate the names of multiple devices.

As previously introduced in section 3.3, the major benefit of using PFQ resides in its programmable fanout described through the PFQ-Lang functional language. As such, the packet fanout mode may indeed be specified by a PFQ-Lang program and placed in the configuration file as in the following example[4]:

```
# Pcap configuration file (PFQ flavor)

def_group  = 11
caplen = 64
rx_slots = 131072

>  main = do
>          tcp
>          steer_flow
```

In some cases, a given group must be associated with a network device rather than a process. This association lets a process handle multiple devices at a time, each under a

[4]Notice the use of the character > to prefix each line according to the *Haskell bird style* as alternative to the fanout keyword

different group of sockets. A typical scenario is that of an OpenFlow Software Switch (e.g., *OFSoftSwitch* [3]), in which multiple instances of the switch can run in parallel using the new `pcap` library, each of them processing a portion of the traffic over a set of network devices.

The `PCAP_GROUP_`*`devname`* environment variable (and its `group_`*`devname`* counterpart keyword in the config file) can be used to override the default group for the process when opening a specific device, as in the following example:

```
PCAP_DEF_GROUP=42 PCAP_GROUP_eth0=11
  tcpdump -n -i pfq:eth0^eth1
```

in which the application *tcpdump* sniffs traffic with the group 11 from device `eth0` and with the default group 42 from the device `eth1`.

Finally, there are cases in which an application needs to open the same device multiple times under different configuration parameters (e.g., with a different criterion for packet steering). In all such cases, the proposed pcap-fanout library provides the concept of *virtual device*, namely a device name postfixed with ':' and a number. This is very similar to the alias device name, but it does not require the user to create network aliases at the system level. As an example, the next two lines allow to collect traffic from the network device `eth0` under two different group (11 and 13) by *virtually* renaming the network interface itself.

```
group_eth0 = 11
group_eth0:1 = 23
```

3.5 Using the pcap fanout in practice

Although the new `pcap` library is ready to use in parallel applications, the attained performance significantly varies according to the overall setup of computation resources.

By design, the code of the library is *re-entrant*, which means that it can be used in any contest, both in a single and in multi-threaded applications. In other words, it just suffices to open a socket, bind it to one or more devices, and join a socket group with a specific fanout algorithm to start receiving a fraction of the incoming traffic.

It goes without saying that a multi-threaded application is expected to open a socket on a per-thread basis and be assigned a dedicated group. This procedure allows it to capture the whole traffic coming from a NIC under the specified packet dispatching. Obviously, the same applies for multiple processes of single-threaded applications. New threads or processes can join a group at any time. In that case, the underlying implementation adapts the fanout stage to deliver packets to a different number of endpoints. While this is a desired feature, it may, however, raise issues about flow consistency. For this reason, applications with strict requirements of flow consistency should restart to guarantee correct results.

Using more than one group is also possible – if supported by the underlying socket – and allows multiple multi-threaded applications (or groups of processes) to receive the traffic coming from the same NIC, possibly under different fanout algorithms and degrees of parallelism.

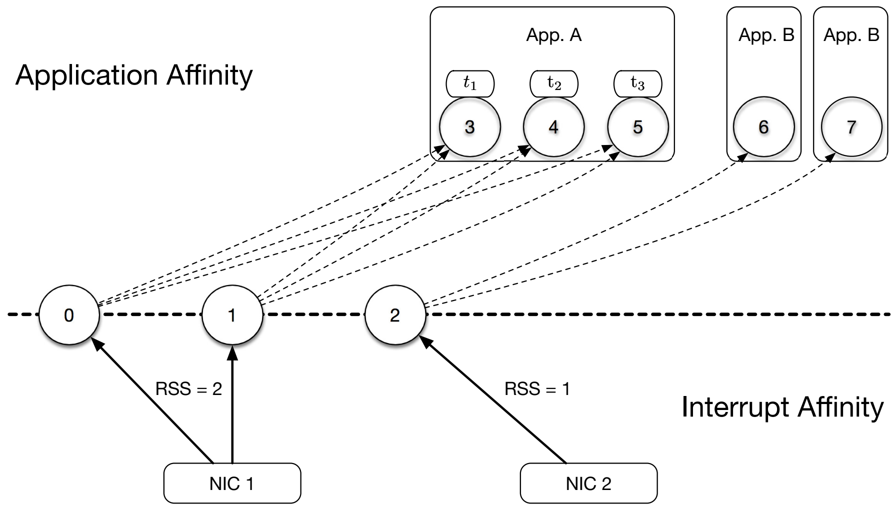


Figure 3.5: *Application and interrupt affinities*

3.5.1 Applications and Interrupt Affinities

When dealing with parallel computation, the first critical issue to be addressed is the configuration of the *application affinity*, namely selection of the set of CPUs that will run the threads (or instances) of the application itself.

Such a scenario is depicted in Figure 3.5 in which, as an example, the affinity of the threads of the application A is set to CPUs 3, 4 and 5 while the affinity of the two instances of application B is set to CPUs 6 and 7. As a good practice, different workers (threads/processes) should run on top of different CPUs to take advantage of maximum computation power. In any case, the application itself is ultimately responsible for setting its affinity. To this aim, the non-POSIX API `pthread_setaffinity_np` as well as the `sched_setaffinity` system call can be used to assign threads and processes to specific cores, respectively.

However, the bottom part of Figure 3.5 shows another critical aspect to be addressed when dealing with packet capturing in general, and in particular with active sockets. In addition to user-defined threads/processes, the Linux operating system provides a dedicated kernel thread for packet capture, called `ksoftirqd`. Such a thread is designed to run on top of any CPU serving the *soft interrupt (IRQ)* scheduled by network cards. All such CPUs running the capturing kernel thread define the so-called *IRQ affinity*.

In recent years, modern NICs (like 10/40G Intel cards) support multiple hardware queues (*channels*), where packets coming from the network are split into using a predefined load balancing scheme – the RSS (Receive Side Scaling) algorithm based on the Toeplitz hash function computed over the packet headers. The distribution of traffic across multiple hardware queues allows multiple `ksoftirqd` threads to fetch network packets in parallel, thus improving the receive performance of the system[5].

Both the number of channels[6] involved in the capturing operations, as well as the specific CPUs selected to serve them, are fundamental parameters to configure for optimal performance. Again, as a good practice, running an application thread and a kernel NAPI context on the same CPU is a very bad idea. Alternatively, wherever possible, the

[5]an analogous behavior occurs in the transmission side

[6]the number of channels is referred to as RSS, where RSS = n implies that n channels are used on that interface

application affinity should be set by avoiding core overlapping with the IRQ affinity.

This latter can be set by using rather naive bash scripts shipped with device drivers code. However, since such an operation involves the configuration of low-level physical parameters, we argue that this should be handled by the `pcap` library by providing a new set of APIs as a handy tool to let the application select the IRQ affinity on-the-fly.

At first, the following APIs deals with device channels:

```
int pcap_set_channels(
        const char *dev,
        struct pcap_channels const * ch,
        int channel_mask,
        char *errbuf);

int pcap_get_channels(
        const char *dev,
        struct pcap_channels *info,
        char *errbuf);
```

These two APIs allow to set and get the number of hardware queues for a given device. In particular, by passing the pcap_channels data structure and the relative channel_mask, the setter function is allowed to selectively update the number of the supported channels, that are *Rx, Tx, Combined* and *Other*. Conversely, the second function is used to retrieve the information about such a number and type of channels enabled for device `dev`. However, depending on the hardware and the driver in use, some channels might not be available. For instance, the Intel 10G card supports combined channels only, and the definition of a different number of Rx and Tx channels is not possible.

Once the information about channels is set, the IRQ affinity can be set/retrieved through the following APIs:

```
int pcap_channel_setaffinity(
        const char *dev,
        int channel_number,
        const cpu_set_t *cpuset);

int pcap_channel_getaffinity(
        const char *dev,
        int channel_number,
        cpu_set_t *cpuset);
```

that allows defining the set of CPUs in charge of handling the IRQs and to retrieve the actual IRQ configuration, respectively.

As an example, the following snippet sets the number of the combined channels to 2 for the device `eth0`:

```
struct pcap_channels ch = \
     { .combined_count = 2 };
if (pcap_set_channels(p, \
   "eth0", &ch, \
   PCAP_COMBINED_CHANNELS) != 1)
{ /* error */ }
```

whereas the statement that follows retrieves the full configuration for the device:

```
pcap_get_channels(p, "eth0", &ch);
```

Analogously, it is possible to specify the IRQ affinity to a specific set of CPUs for every single channel. The following example sets the affinity for the channel 0 to core 0 and channel 1 to core 1, respectively:

```
cpu_set_t cpuset;
CPU_ZERO(&cpuset); CPU_SET(0, &cpuset);
if (pcap_channel_setaffinity(
   "eth0",
   0,
   &cpuset) != 1) { /* error */ }

CPU_ZERO(&cpuset);
CPU_SET(1, &cpuset);
if (pcap_channel_setaffinity(
   "eth0",
   1,
   &cpuset) != 1) { /* error */ }
```

Finally, notice that the whole procedure can also be replicated by declaring a few keys in the configuration file, as in the example reported next:

```
combined_channels@eth0 = 2
irq@eth0.0 = 0
irq@eth0.1 = 1
```

3.6 Performance Evaluation

This section aims at assessing the performance of a simple multi-threaded application using the new `pcap` library through the extended API when running on top of the standard Linux socket as well as PF_RING and PFQ.

The experimental test bed consists of a pair of identical PCs with an 8-core Intel Xeon E5-1660V3 on board running at 3.0GHz and equipped with Intel 82599 10G NICs and used for traffic capturing and generation, respectively. Both systems run a Linux Debian distribution with kernel version 4.9.

3.6.1 Speed-Tests

The first set of tests aims at assessing the impact of fanout in the performance of the lightweight multi-threaded `pcap` application `captop`[7] that simply counts the received packets when running on top of the standard Linux socket, PF_RING and PFQ, under different packet sizes and number of underlying capturing cores (different RSS values). Packets are synthetically generated at 10 Gbps full line rate by `pfq-gen`, an open-source tool included in the PFQ distribution.

Figure 3.6 shows the result of the speed-test when *four working threads* of `captop` retrieve the packet streams on top of the `TPACKET` Linux socket according to different fanout modes. The whole set of measurements is replicated by progressively increasing

[7]Available at https://github.com/awgn/captop

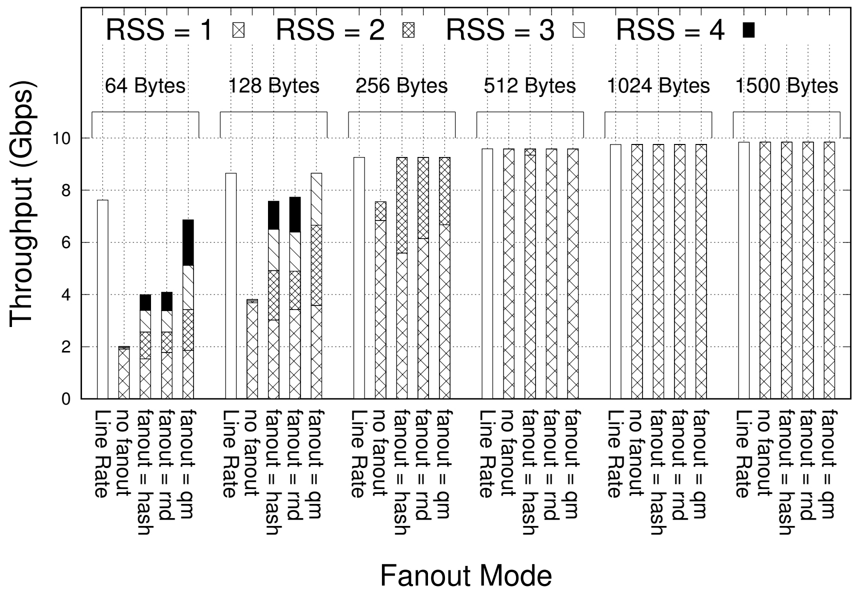


Figure 3.6: *10 Gbps packet capture with* `libpcap` *over standard Linux socket*

the number of underlying capturing cores, from 1 to 4 (RSS = 1,. . . ,4), yet keeping the application and interrupt affinities not overlapped. Moreover, as a reference value, the theoretical line rate limit as well as the capturing rate of a *single-threaded* instance of `captop` ("no fanout") are also reported for each packet size.

The performance figures are in line with the expected capabilities of the `TPACKET` socket and show that full capture rate is reached at around 128 Bytes long packets when using the lightest "qm" dispatching algorithm and at the 256 Bytes long packets in all other cases. However, further interesting insights come out from the figure. Indeed, especially for short packets, the introduction of fan–out turns out to accelerate the overall application capture rate. This effect was somewhat unexpected, as fan–out is used to distribute traffic among up–layers working threads and should not impact the pure underlying capture rate. In fact, this beneficial effect is likely due to the internal implementation of the Linux socket that proves to be inefficient in handling contentions when multiple cores (i.e., `NAPI` contexts) concurrently inject packets to a single socket (or to memory mapped rings in the case of `TPACKET`). With fan–out enabled, when the number of application sockets increases, the contention on the socket queues among multiple napi–threads is reduced accordingly, and this determines a beneficial impact on the performance. In addition, the observed performance acceleration varies with the fan–out mode, as each algorithm has a different computational cost. As a result, the lightest "qm" fan–out mode (that simply matches an integer number), proves to outperform both the "rnd" and "hash" modes which need to either generate random numbers or compute hashes functions before dispatching packets to the target sockets.

Figure 3.7 shows the results of the same test when the default Linux socket is replaced by PF_RING using the "flow" and the "round_robin" fanout schemes. As expected, the use of vanilla drivers does not allow to attain stellar performance which, for small packet sizes, drops below the ones reached by the standard socket. We deem that much better figures could be reached by using the set of "aware" drivers once shipped

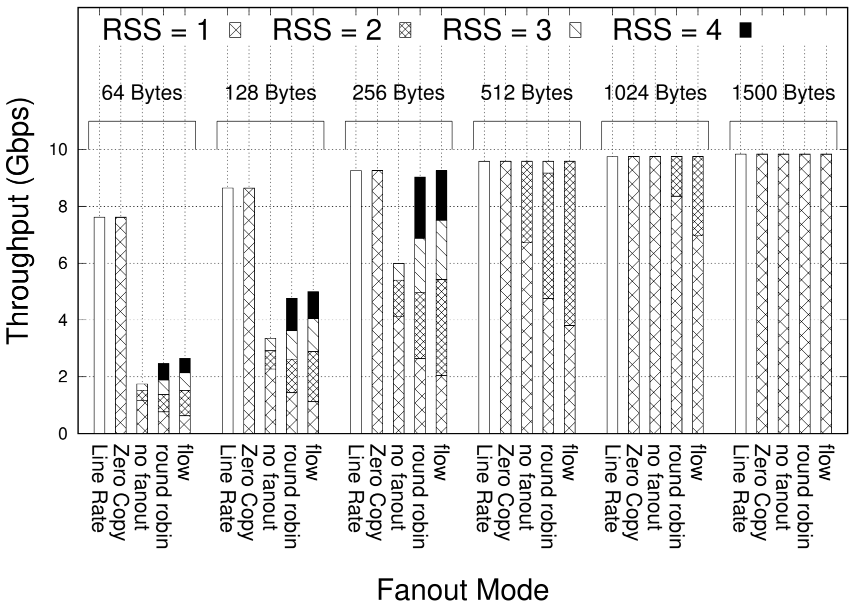


Figure 3.7: *10 Gbps packet capture with* `libpcap` *over PF_RING socket*

with the PF_RING release.

However, even in this case, line rate packet capture is still reached for data length of 256 Bytes and the same beneficial effect of packet fanout on the socket capture performance is observed.

For reference purposes, Figure 3.7 includes the performance of the passive socket PF_RING Zero Copy (ZC), running underneath the new pcap library (to which it keeps semantic compliance), although with no fan–out support. Again, the results are in line with the expected capture potential of the socket that proves to attain line rate speed on a single capturing core. However, notice that packet distribution over multiple application threads would not be possible in this case unless the commercial ZC library is purchased separately. Without such a library, the application threads can be increased through the RSS algorithm only, albeit they cannot be decoupled from those fetching packets from the interface.

Figure 3.8 shows the results of the same test when the PFQ socket is used to capture packets and distribute them according to analogous fan–out modes (steering algorithms). Again, the performance of the `pcap` application is consistent with the typical PFQ capture figures which prove to reach line rate speed even with the shortest packet size. However, in this case the use of fan–out does not accelerate the application performance. This, in fact, is the expected effect of fan–out and it is consistently observed as the internal lock–free queues of PFQ manages multi–core access contention efficiently.

The last experiment of this section aims at showing the impact of the correct selection of the application affinity on the capture performance. In this test, four capturing cores (RSS = 4) retrieve a full 10 Gbps stream of traffic from the physical device and distribute the workload according to the "qm" fan–out mode across four working threads of `captop` that counts the number of received packets and bytes. The four threads of the application are allowed to run on top of the capturing cores, so as to span the full range of overlapping configurations. Figure 3.9 shows the capture speed

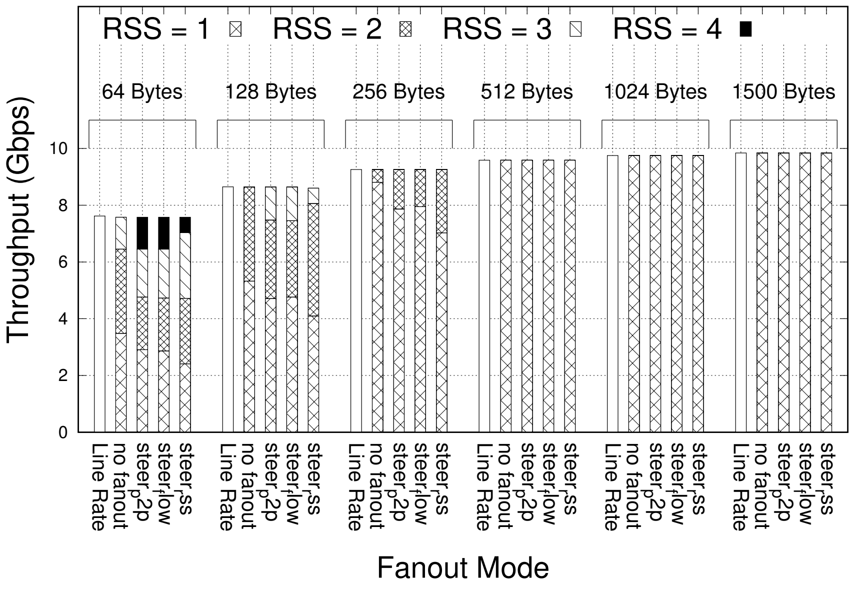


Figure 3.8: *10 Gbps packet capture with* `libpcap` *over PFQ accelerated socket*

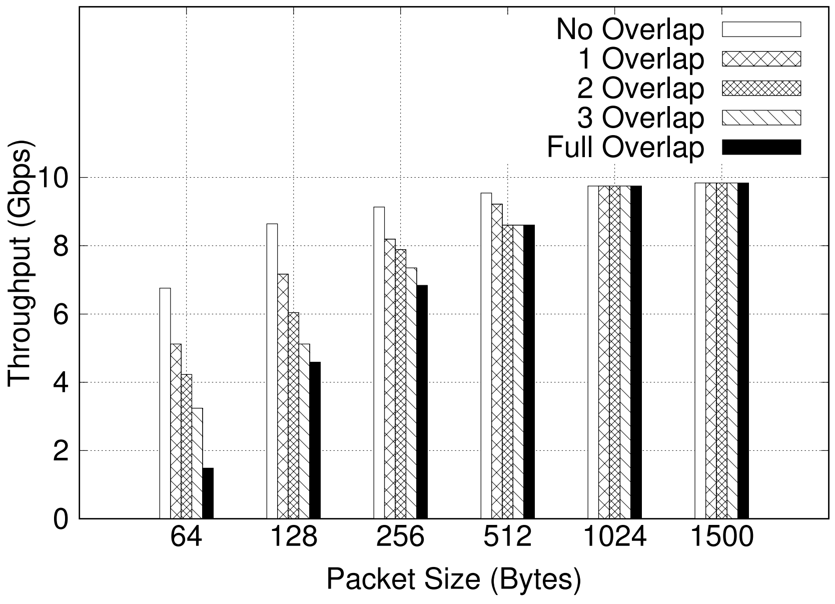


Figure 3.9: *Capture speed vs. application irq affinity*

attained when the number of overlapping cores increases from zero (best affinity configuration) to four (worst affinity configuration) for different packet sizes. As expected, in spite of the light computational burden of `captop`, the capture performance significantly degrades when the affinity of application threads and interrupts overlap. This effect is well visible for short packet length, and vanishes when the packet size increases due to the lighter effort required to the capturing cores.

3.7 Use-cases

In this section the performance of the new `pcap` library in practical use-cases is presented. To this aim, the two well-known network applications *Tstat* and *Bro* have been selected as they are both single-threaded and support live traffic access through the `libpcap` library.

In the following experiments, *Tstat* and *Bro* are flooded with different traffic streams at 10 Gbps speed. Synthetic and real (VoIP) UDP traces with different mean packet sizes are used in the *Tstat* experiments, while a real packet trace containing both TCP and UDP traffic is used with *Bro*. As will be elaborated upon, in some cases the fanout alone allows to scale-up the processing power up to full rate capacity while, in another case, socket acceleration must be enabled to attain top performance figures. In all tests, the following metrics are observed:

- *Link received*, the number of packets captured and managed by the socket. Later, it will be represented as a fraction of the packets that are transmitted by the traffic generator;
- *IF dropped*, the number of packets that cannot be handled by the socket and are dropped at the interface level. Notice that the sum of *IF dropped* and *Link received* is the total number of packets sent;
- *App. received*, the number of packets processed by the application, represented as a fraction of the packet received at the socket level (*Link received*);
- *App. dropped*, the number of packets dropped because the application is backlogged. Again, notice that *App. received* + *App. dropped* = *Link received*.

The first two metrics reflect the socket capture efficiency, and can only be improved by utilizing socket acceleration. Conversely, the remaining metrics are associated with the application processing speed and can be improved by enabling packet fanout.

In all experiments, both *Tstat* and *Bro* were run with their default configurations as the primary purpose was to show how performance scales up with multiple cores rather than focusing on any specific application setup.

3.7.1 Tstat

Tstat [9] is a widely popular tool for generic traffic analysis. It includes a large number of deterministic and statistical algorithms and can be used for the post-processing of trace files as well as for stream analysis of live data using the `pcap` library.

In the first experiment, *Tstat* runs on top of the standard Linux socket (configured with RSS=3) and is injected with synthetic UDP traffic with an average packet size of 300 Bytes containing up to 4096 different flows. The input traffic rate saturates the full 10 Gbps line speed, with an average packet rate of 3.8 Mpps. The results are shown in Figure 3.10 and prove that while the Linux socket catches up with the input traffic speed, a single instance of the application does not, on our hardware. However, by simply enabling packet fanout, two working instances of *Tstat* are sufficient to process all the received packets.

When TPACKET is replaced by PF_RING, the overall performance is somewhat similar, with two application instances capable of processing most of the offered traffic.

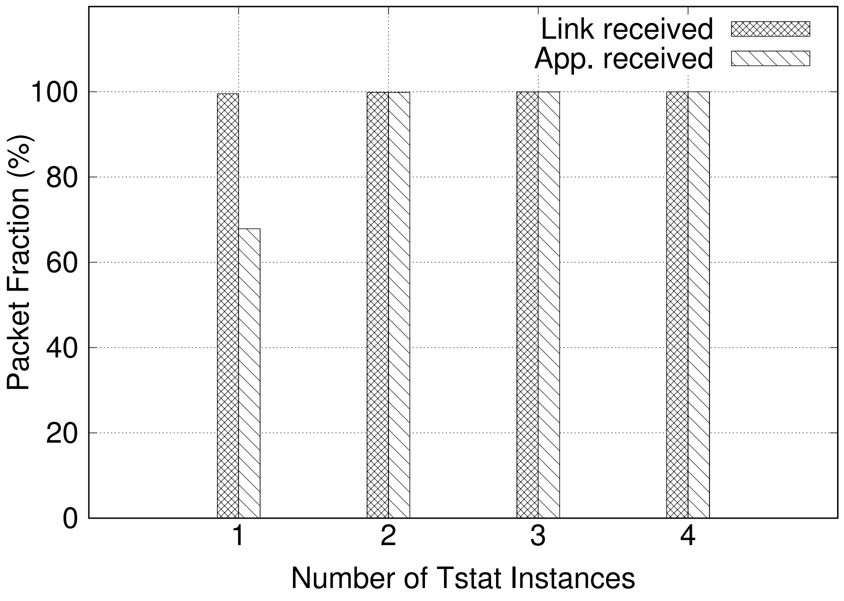


Figure 3.10: *Tstat and Linux socket: 10 Gbps traffic analysis with 300 Bytes average packet size*

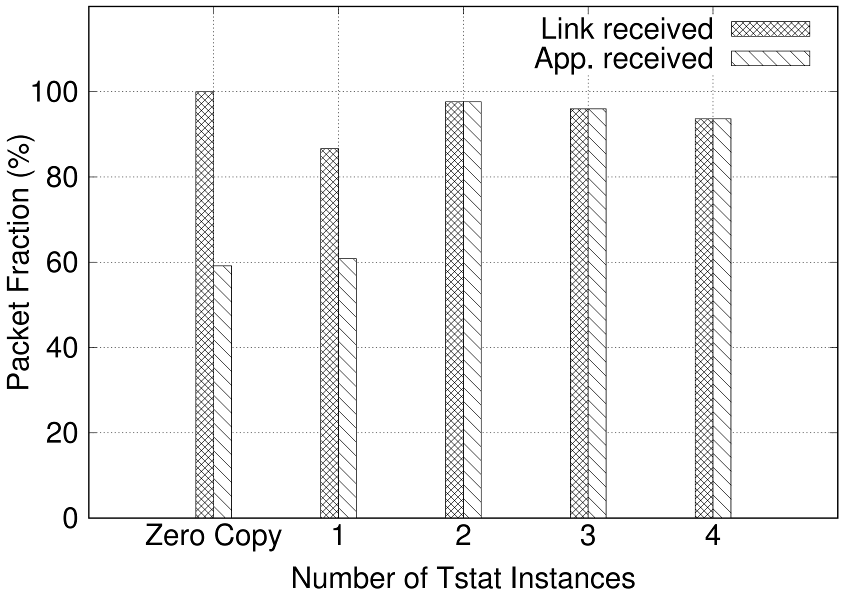


Figure 3.11: *Tstat and PF_RING socket: 10 Gbps traffic analysis with 300 Bytes average packet size*

The figure also shows some fluctuations when using more than two *TSTAT* workers which may likely be due to contentions that occur when the three capturing kernel threads push packets in the application socket queues. Again, the results of PF_RING ZC are also reported and prove that, despite being able to capture the full amount of traffic stream, the spare amount of processing resources available on the core used to fetch data is not enough to allow the application to process all the received packets.

Figures 3.12 and 3.13 report the results of TPACKET and PF_RING when running the same experiment with average packet size decreased up to 128 Bytes (and corre-

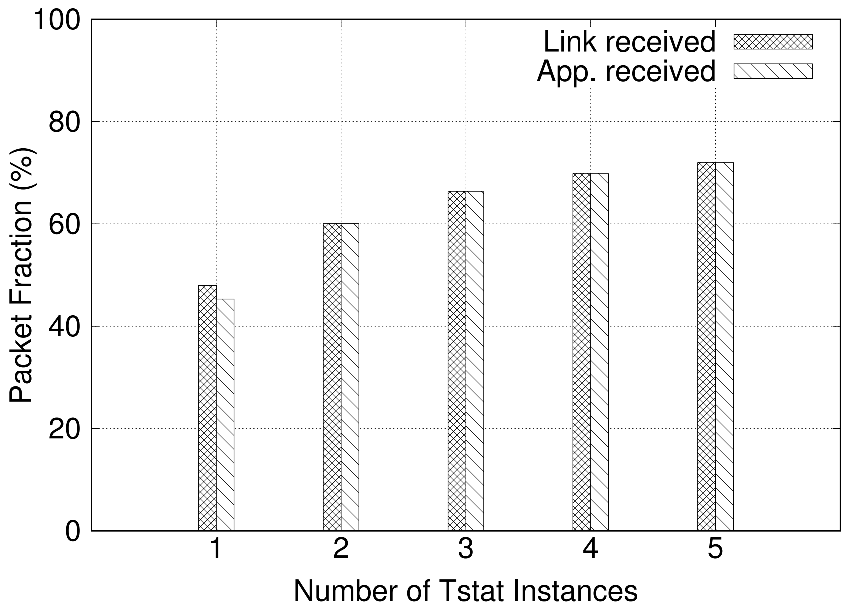


Figure 3.12: *Tstat and Linux socket: 10 Gbps traffic analysis with 128 Bytes average packet size*

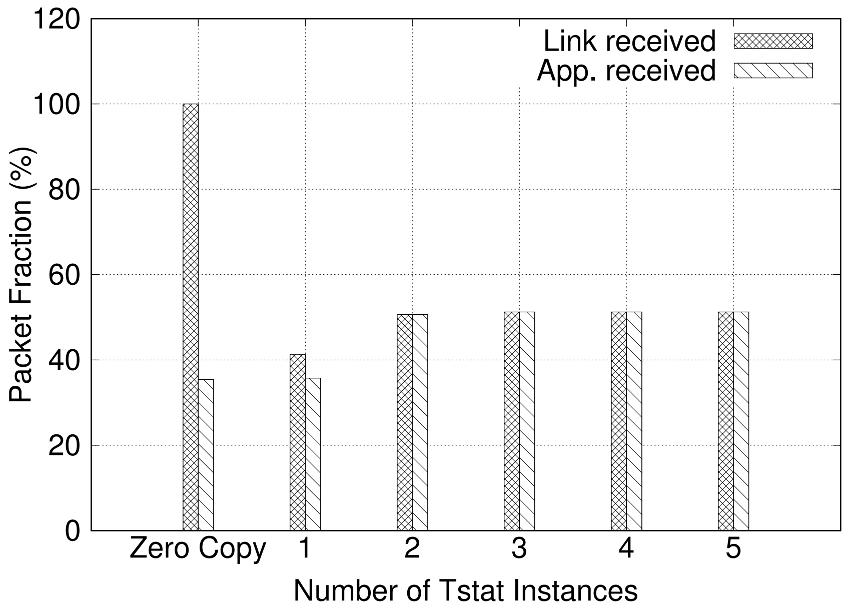


Figure 3.13: *Tstat and PF_RING socket: 10 Gbps traffic analysis with 128 Bytes average packet size*

sponding average packet rate pushed up to 8.2 Mpps). In both cases, the use of fanout allows two working instances of *Tstat* to effectively process all the packets received on the physical device. However, nearly 40% and 50% of the input packets turn out to be dropped at the network interface as the input traffic rate exceeds the potential capture rates of the `TPACKET` and `PF_RING` socket, respectively. Also, it can be noticed that the fraction of data processed by the application is even lower in case of using PF_RING ZC, due to the higher CPU consumption required by the underlying packet capturing operations that run on the same CPU.

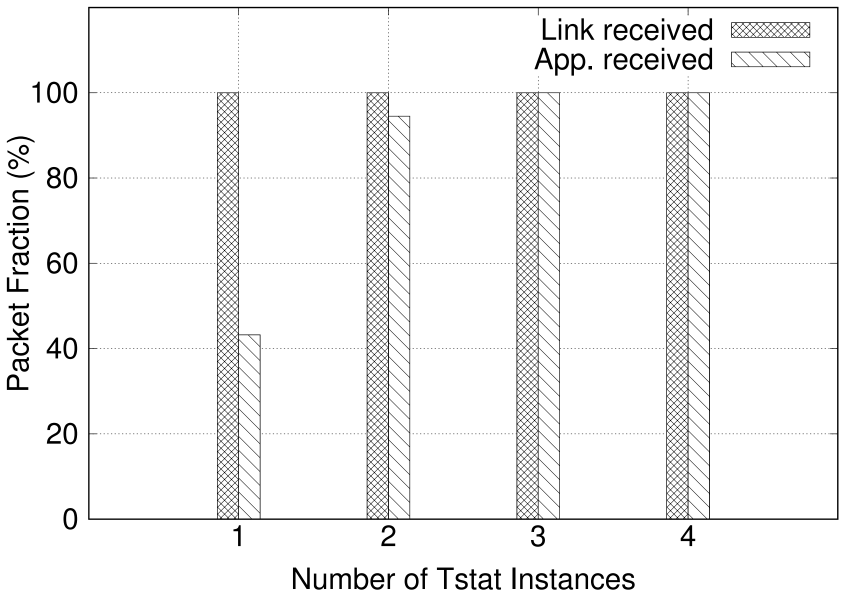


Figure 3.14: *Tstat and PFQ: 10 Gbps traffic analysis with 128 Bytes average packet size*

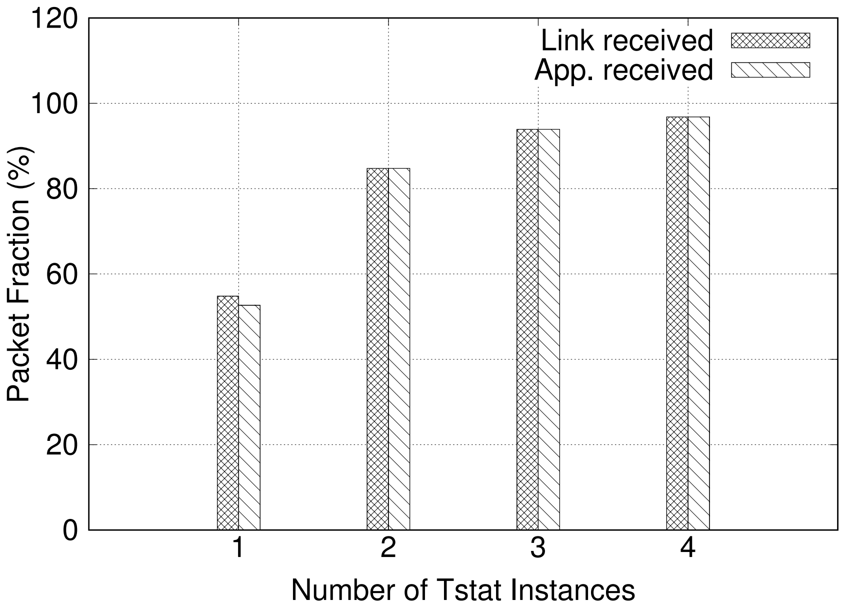


Figure 3.15: *Tstat and Linux socket: Real VoIP trace traffic analysis*

To improve the performance of the application, packet fanout can conveniently be combined with underlying socket acceleration. Indeed, as shown in Figure 3.14, the use of PFQ allows to avoid packet drop at the lower level and packet fanout allows three instances of *Tstat* to successfully process nearly all of the input traffic.

To test the new library performance in a more realistic scenario, we run a further set of experiments in which *TSTAT* is called to process a real time traffic trace that contains around 200K VoIP flows collected over a backbone network. The performance was recorded by feeding a varying number of *TSTAT* instances with the VoIP packet data at

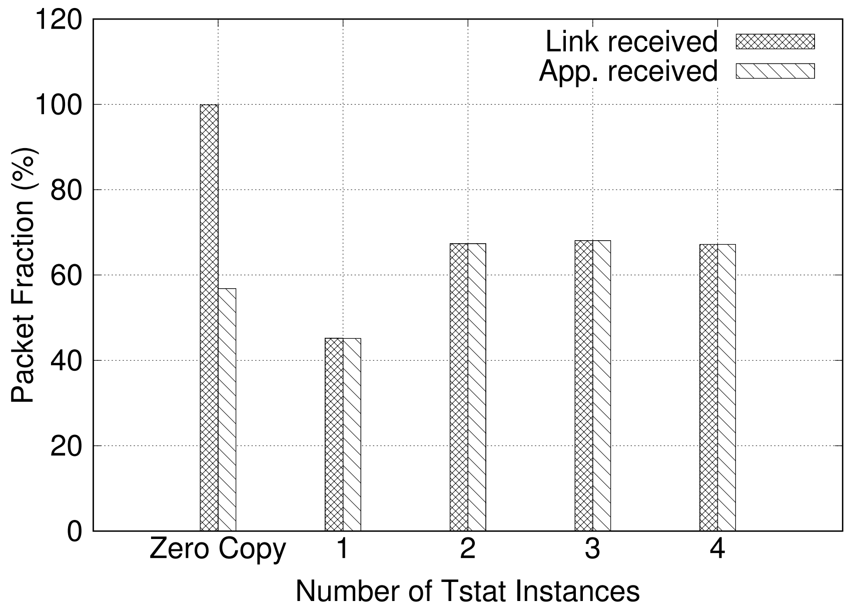


Figure 3.16: *Tstat and PF_RING socket: Real VoIP trace traffic analysis*

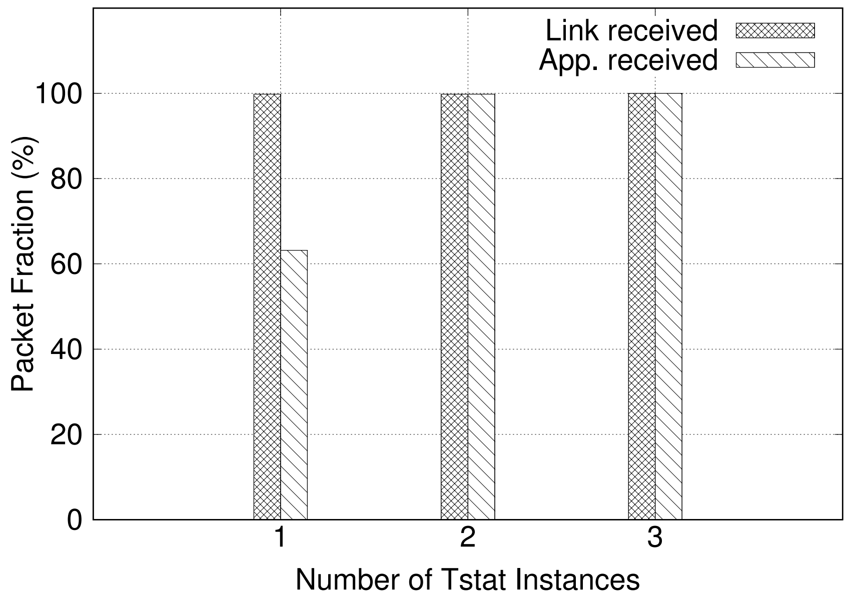


Figure 3.17: *Tstat and PFQ: Real VoIP trace traffic analysis*

around 5 Gbps speed.

Figures 3.15, 3.16, 3.17 show the results of the three experiments when using TPACKET, PF_RING and PFQ capture sockets, respectively. In the first two cases, the underlying number of channels was set to 4 (RSS=4) to get the best possible performance, whereas only two channels (RSS=2) were sufficient to fetch all packets from the network device when using PFQ.

Overall, the results confirm the findings of the tests previously carried out with synthetic traces. The fan-out feature provided by the `pcap` library allows TPACKET to scale its performance and let four *TSTAT* instances process nearly all of the traffic. Notice that the number of instances could not be increased without colliding with the underlying IRQ affinity – as our architecture is equipped with only eight cores.

Conversely, PF_RING still exhibits performance saturation up to the second fetching cores, so increasing the number of *TSTAT* instances beyond two does not increase the percentage of packets processed by the application. Consistently, the single instance of *TSTAT* running on top of PF_RING ZC does not reach 60% of the overall amount of packets fetched by the socket itself.

Finally, the PFQ socket allows the `pcap` library to effectively distribute traffic across the applications instances so starting from two instances of *TSTAT* is sufficient to process all of the incoming traffic with two CPUs only (RSS = 2) set to run packet fetching threads in the kernel space.

3.7.2 Bro

Analogous tests have been carried out to assess the performance of the *Bro* network security monitor [1] running on top of the new `pcap` library.

Bro is a single-threaded computation intensive application that can be run in both standalone and cluster configuration. In the second case, the total workload is spread out to multiple instances (nodes) across many cores by a *frontend*. Messages and logs generated by all nodes are then collected and synchronized by the *broctl* manager to provide a unified output.

To date, the classic `pcap` library could only be used in the single node configuration. Indeed, to enable parallelism in the cluster deployment, additional on-host load balancing plugins are required (currently, available plugins are available for PF_RING and Netmap sockets). The introduction of packet fanout, instead, enables the use of the `libpcap` interfaces even in the cluster configuration by only setting a few environment variables without the need for extra plugins.

In the next experiments, a cluster of *Bro* nodes using the new `pcap` library is fed with a real packet trace played at 2.4 Mpps, corresponding to full 10 Gbps line speed. The trace was collected over a multi-gigabit link and contained an aggregate of a few thousand of TCP and UDP flows. However, given the lower PF_RING scaling capability, only the TPACKETS and the PFQ sockets were used.

Due to the high computation demand requested by each node, CPU hyperthreading technology was enabled when the number of *Bro* instances exceeded the number of physical cores.

Figure 3.18 shows the cluster performance when the standard Linux socket was used with two underlying capturing cores (RSS=2). The beneficial effect of fanout is visible as the fraction of packets received by the application scales up to the whole amount of packets received by the socket. However, the fraction of packet dropped at the interface is quite relevant (up to 40%) and raises the need for socket acceleration.

Indeed, Figure 3.19 shows the results obtained when the standard socket is replaced by PFQ under the same number (two) of capturing cores. The use of the accelerated socket dramatically reduces the packet drop rate at the interface up to negligible values. This, in turn, significantly increases the number of packets available to the working nodes whose performance, indeed, scales linearly up to seven *Bro* instances. With more than seven sockets the fraction of packets received by the application still increases lin-

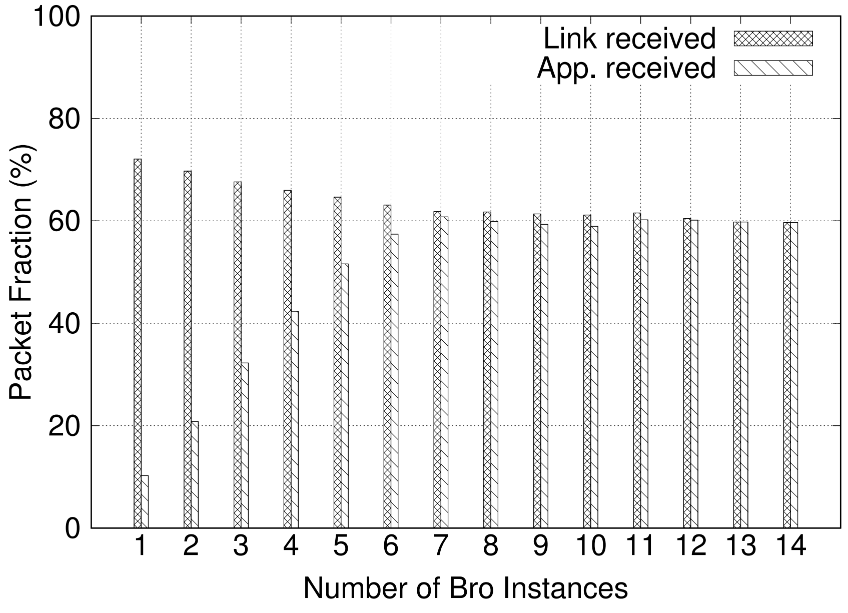


Figure 3.18: *Bro: real traffic analysis with standard Linux socket*

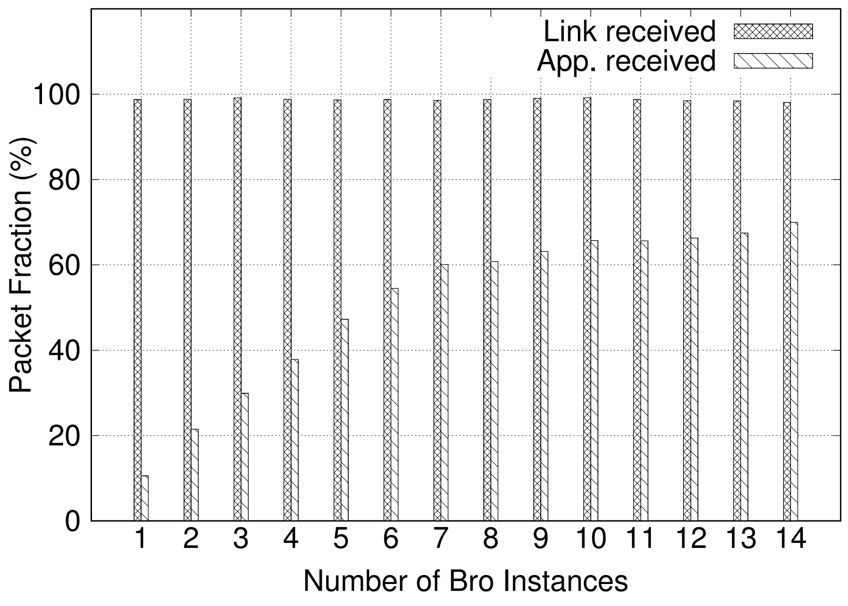


Figure 3.19: *Bro: real traffic analysis with PFQ*

early, but the slope is reduced as the additional cores available through the hyperthreading technology do not have the same computational power of physical CPUs. Finally, notice that the number of physical cores of the PCs used in the experimental setup limited the maximum cluster cardinality to 14 nodes, as two of the overall 16 available cores were dedicated to underlying capturing/steering operations.

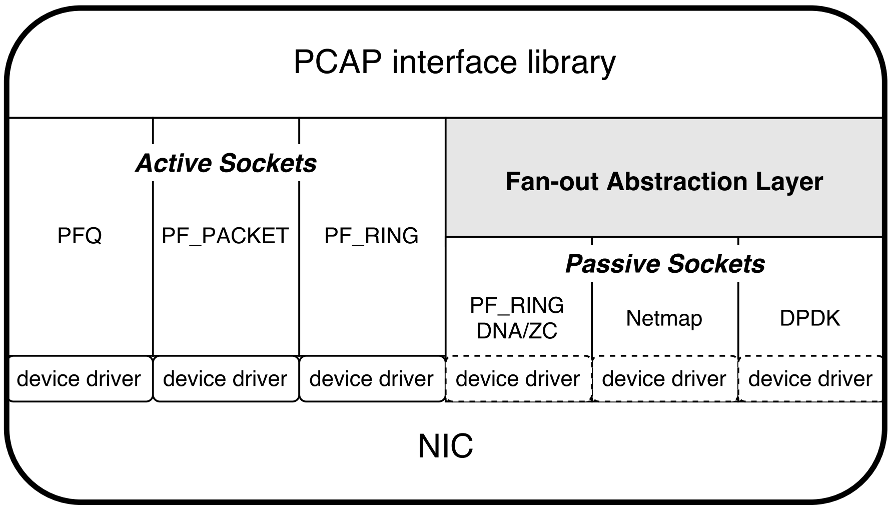


Figure 3.20: *The fanout Abstraction Layer*

3.8 Towards the integration of passive sockets

In section 3.3 we have discussed the complexity of providing packet fanout support for passive sockets, such as netmap and DPDK. This section is meant to further elaborate upon this topic and to provide a possible unified architecture to include the support of passive sockets within the `pcap` library with fan-out feature. Figure 3.20 depicts the complete scheme of the `pcap` library as we envision it. The `pcap` interface stands in the top part and is made of a set of functions that are invoked by the user-space applications to manage sockets and receive/injects packets to the network. In fact, such functions are indeed *virtual functions* (i.e., function pointers), and their actual implementations are provided by the underlying blocks. Under the hood, the two families of passive and active sockets require different management. Since passive sockets lack active threads that fetch packets from the NIC, it is necessary to build an *abstract layer* that can handle traffic data and apply the fanout directives. We named this layer as *fanout Abstraction Layer* (FAL). Currently, the blocks on the left-hand side of the figure (i.e., the ones associated with the active sockets PFQ, PF_RING, and TPACKET) are fully implemented. On the right-hand side, instead, the fanout abstraction layer is still under development, and its design is presented here as an ongoing research activity. In short, the role of the FAL is to hide the underlying machinery of passive sockets by exposing to the upper layer a *abstract active socket* (the `fal` socket itself) that can be accessed and managed by applications through the `pcap` library with no specific modification to their the source code. As such, the FAL layer is responsible for translating the virtual directives of FAL socket into real operations made onto actual sockets (netmap, DPDK, etc.) and consists of *active poller processes* that fetch packets from the network interfaces, apply the requested packet fanout algorithm, and finally deliver packets to the applications.

Under the above assumptions, the minimal set of API exposed by the FAL includes four basic classes of functions for:

- opening/closing the FAL socket (`fal_open`, `fal_close`);

- managing fanout groups and algorithms (`fal_fanout`, `fal_join`, `fal_leave`);
- attaching/detaching the `FAL` socket to physical network devices (`fal_bind`, `fal_unbind`);
- implement classical I/O primitives for receiving and transmitting packets (`fal_recv`, `fal_send`, `fal_dispatch`);

Finally, as the system is intended to support both threads and processes, the implementation of FAL is designed to store the configuration data in a shared memory, possibly accommodated on top of the Linux HugePages for performance reasons.

CHAPTER 4

Functional Packet Processing

4.1 Introduction

The central role of the Internet as the key infrastructure driving global social and economic processes makes it the perfect ecosystem for cyber-threats and network attacks to mutate and become more and more “effective” toward their malicious purposes.

In such a scenario, network nodes – once limited to switches and routers, in charge of precise operations – see their mission significantly extended and their functions complemented by the adoption of heterogeneous *middleboxes* providing specific services such as intrusion detection systems, firewalls, address translation, etc. However, the appearance of such new devices targeted at very specific functions brings several drawbacks. At first, middle-boxes are typically proprietary and based on closed source software which makes interoperability and management in large-scale multi-vendor scenarios a big issue. Also, their flexibility and programmability are generally pretty limited, and the programming skills required to customize and configure their operations largely exceed those of average network administrators. As a result, device programmability gets significantly impaired and chances for innovation and experimentation reduced.

So far, Openflow [74] is the most effective and convincing answer to the above-summarized issues. Its pragmatic approach of proposing an open and stateless programming interface to (closed) inner functions of switching nodes was the key to convince device manufacturers (naturally reluctant to disclose their low-level implementation) to expose a virtually vendor-neutral API to enable a reasonable degree of programmability of network devices. As a result, OpenFlow is today the leading platform to develop Software Defined Networking (SDN) solutions.

This chapter presents an analogous (software defined) approach to the development of generic middleboxes whose behavior can be entirely “defined” by an open programmable interface equipped with an easy, expressive and robust programming language.

The system herein described consists – at the low level – of a high performing software platform implementing a wide variety of primitive functions that exposes – at the higher level – an interface that can be programmed by a specific functional language.

Why a functional language?

The selection of an adequate programming model for the above-introduced system is not of secondary importance. At first glance, the choice of a functional model may look like an unnecessary overkill as it introduces the need to adhere to a strict formalism as opposed to the pretty loose constraints posed by traditional *imperative* languages. Such strict requirements, however, turn out to significantly ease the programming interface, while providing the user code with the typical robustness of functional programming. Indeed, as it will be shown in the rest of the chapter, typical packet processing operations will be instantiated by writing *a few lines of code* only, in which the chain of packet processing is represented by the composition of elementary operations. As such operations may be executed in parallel, the *immutability* of data enforced by the functional paradigm prevents possible race conditions. Furthermore, before being executed in the underlying processing engine, the user code is formally verified against the strongly typed language and enforced by the functional model. However, it is worth mentioning that middlebox programmers are not required to know "gory" details of the underlying data-plane implementation as such details are entirely hidden behind the set of instructions exposed by the functional language. Naturally, skilled users may still extend the number or modify the behavior of the data-plane primitives included in the system.

The contribution

In practice, the ideal positions expressed so far have been translated into the design and development of an open platform for the development of a generic network middlebox on the Linux platform. To this aim, a functional engine implementing generic data-plane primitives has been integrated on top of PFQ [21], the packet capturing platform presented in chapter 2. The execution of the actual operations performed in the underlying engine is controlled by programming a functional interface implemented as a Domain Specific Language (DSL). As a result, the device (middlebox) gets its computation machinery decoupled from its (programmable) control logic, hence enabling a software-defined approach to packet processing.

The chapter is organized in a top-down fashion, by introducing at first the functional model (Section 4.2) for packet processing and its theoretical foundation rooted in the definition of *monads* (Section 4.3). Section 4.4 presents pfq-lang, the specific functional language developed to program the middlebox behavior, by giving an overview of the available functions. Section 4.5 delves into the lower level system implementation with more details on the system's components. Section 4.6 presents the performance of the overall system while Section 4.7 shows a set of simple real use cases, including monitoring applications, firewalls, legacy applications, and so on.

4.2 Functional Packet Processing

The rationale behind this study comes from the simple intuition that network applications can be modeled as a packet processing pipeline composed of a sequence of ele-

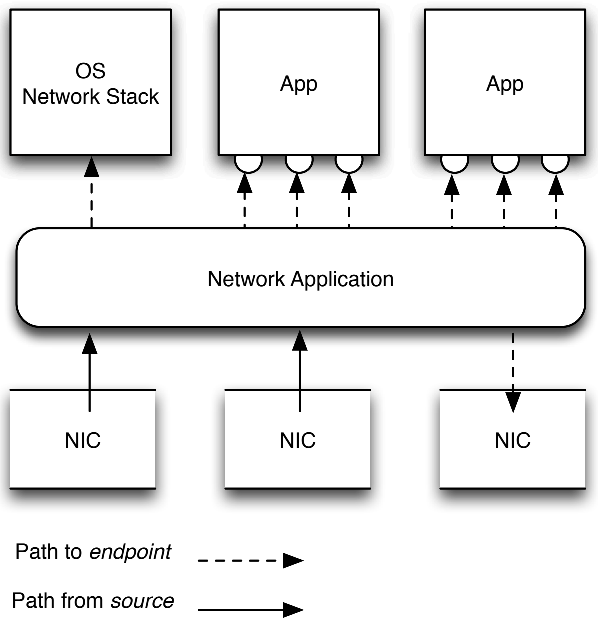


Figure 4.1: *Logical scheme for network applications*

mentary operations that consume data (packets) and produce computations, i.e., packets associated with actions (e.g., forwarding, storing, filtering, etc.).

In this logical scheme (see Figure 4.1), packets are retrieved from one or more *sources* (for example, one or more packet queues from a Network Interface Card – NIC), processed through the pipeline and finally delivered to one or more *endpoints*. Possible endpoints can be either network devices to forward packets, or the sockets of other applications in charge of performing additional processing (e.g., second-stage applications, GPU accelerated computations, etc.), or the networking stack of the operating system itself for ordinary operations. Along with their way to the destination endpoints, packets can be subjected to a full gamma of operations that may produce different packets (e.g., NAT, TTL decrement), add annotations on the packets themselves (e.g., marking as a result of classification), and select the endpoint(s) for delivery.

More in detail, three big classes of operations can be identified within the processing pipeline:

- operations that compute the final endpoints (and their delivery mode) – i.e., the packet *fanout*;
- operations that annotate meta-data on packets (by updating a *state*);
- operations that log data associated with the packet and possibly generate I/O.

From a functional programming point of view, all these operations (that will hereafter be referred to as *actions*) can be modeled by using a new data type `Action P` built around the data type `P` used to represent a packet. The goal of this chapter is to model the elementary operations that compose the packet processing pipeline as *pure* functions (that is, functions without side effects) f mapping packets (with domain `P`) into actions (with domain `Action P`):

```
f: P → Action P
```

As previously mentioned, the first class of operations to be modeled is the one that defines the fanout of a packet. The fanout is represented by the endpoints (if any) to which packets are to be delivered, as well as the way this is accomplished.

The elementary fanout operations are:

- `Broadcast`, which delivers copies of the packet to all the endpoints;
- `Deliver class`, which deterministically delivers copies of the packet to a subset of endpoints specified by the `class` parameter;
- `Steer hash`, which *randomly* delivers the packet to a picked endpoint out of all endpoints based on a property of the packet (e.g. symmetric `hash` of the canonical 5-tuple);
- `Dispatch class hash`, which *randomly* delivers the packet to a picked endpoint out of the `class` subset of endpoints based on a property of the packet (e.g. symmetric `hash` of the canonical 5-tuple);
- `Pass`, which passes the packet to the next processing stage without specifying any endpoint;
- `Drop`, which stops the packet processing, and does not deliver it to any endpoint.

The different packet dispatching modes respond to the possible requirements of multi-threaded network applications as well as to the wide heterogeneity of network protocols. In such a framework, the availability of a fine-grained mechanism to handle parallelism among threads and network devices is crucial to take advantage of powerful multi-core architectures.

However, at first glance, these operations do not seem to be easily modellable as pure functions. For example, the fanout operation `Drop`, which interrupts the processing pipeline, is hard to implement in a purely functional framework. The definition of a new data type constructor `Fanout` helps in addressing this kind of issues.

Definition 4.2.1. *Let the* `Fanout` *type constructor*[1] *be:*

```
TypeDef Fanout P = Drop | Pass P |
                   Broadcast P |
                   Deliver Class P |
                   Steer Hash P |
                   Dispatch Class Hash P
```

where **TypeDef** *is used here to define a new type, while* `Hash` *and* `Class` *are types representing the hash value and the class (i.e. subset of endpoints) used to fully specify the fanout.*

[1] Notice that the Haskell syntax uses a different construct (**data**) and requires the parameter following the type constructor Fanout to be a *type variable*, hence written as a lowercase letter.

This definition indicates that `Fanout P` is a new data type, parametrized by the `P` data type, and enriched by the dispatching modes mentioned above. In particular, the `Deliver` and `Steer` constructors have additional arguments that specify their behavior: `Deliver` takes a mask of type `Class` that identifies a subset of endpoints, whereas `Steer` takes a value of type `Hash` to select a single endpoint (e.g. through a folding or modulo operation). Finally, `Dispatch` takes two more parameters, one of type `Class` and one of type `Hash`, and combines the `Deliver` and `Steer` fanout.

Along with the newly defined data type, two functions – `unit` and `compose` - can be used to compute a fanout operation (generally called "computation") from a packet and to compose two different functions that associate operations to packets (notice that `compose` is needed because the functions have different domains and codomains). The prototypes of `unit` and `compose` are:

```
unit:    P → Fanout P
compose: (P → Fanout P) x (P → Fanout P) →
         (P → Fanout P)
```

meaning that `unit` maps packets (of type `P`) into computations (of type `Fanout P`), while `compose` maps pair of functions from packets to computations into functions from packets to computations.

Definition 4.2.2. *The* `unit` *function is defined as:*

```
unit p = Pass p
```

The `compose` function is more difficult to describe, and can be better formulated in terms of a function indicated as "$\star$" that, given a computation and a function from packets to computations generates a fanout computation:

```
⋆: Fanout P x (P → Fanout P) → Fanout P
```

Notice that the domain of $\star$ is `(Fanout P) x (P → Fanout P)`, which represents the set of possible pairs (computation, function from packets to computations), as the `x` symbol represents the Cartesian product. In the functional programming community, such a function (known as *bind*) is often declared as:

```
⋆: Fanout P → ((P → Fanout P) → Fanout P)
```

by using the so called "*currying*": a function $f : A \times B \to C$ having two arguments (in the sets A and B) and returning a value in set C is equivalent to a function $f_{curry} : A \to (B \to C)$ having only an argument in set A and returning a function from set B to set C. However, for the sake of simplicity, in this chapter, we use the more straightforward notation based on functions with multiple arguments and the Cartesian product.

Definition 4.2.3. *The $\star$ function is defined as:*

```
a * f = case a of
        Drop        → Drop
        Pass p      → f p
        Broadcast p → case f p of
                      Pass p → Broadcast p
                      otherwise → f p
        Deliver c p → case f p of
                      Pass p → Deliver c p
                      otherwise → f p
        Steer h p → case f p of
                      Pass p → Steer h p
                      otherwise → f p
        Dispatch c h p → case f p of
                      Pass p → Dispatch c h p
                      otherwise → f p
```

According to this definition, $\star$ works as follows:

- if a `Drop` operation is combined with any kind of function using $\star$, the result is always `Drop`. This means that once a function drops a packet, the following functions in the pipeline are not evaluated, because the final result of the pipeline is `Drop` anyway;
- if a `Pass p` operation is combined with a function, the packet `p` is passed as an input to such a function, which generates the resulting computation;
- if a `Broadcast p`, `Steer h p`, `Dispatch c h p` or `Deliver c p` operation is combined with a function, the function is applied to the packet `p`. The function can select a new operation for the resulting packet, or can simply keep the input operation (if the result of the function is `Pass p`).

Based on $\star$, the `compose` function can be defined as

```
(f1 compose f2) p = (f1 p) * f2
```

The presented `Fanout` data type constructor allows to model operations to be performed on packets (broadcast, drop, steer, etc.). However, a real packet processing pipeline has also to perform more complex operations, such as marking a packet, delivery of a packet to a NIC, associating some information with a packet and so forth. This can be achieved by extending the `Fanout` type constructor and adding some notion of *state*, to allow the pipeline stages modify such a state. To do this, a more complex data type constructor, named `Action`, has to be defined. The complete `Action` datatype constructor, which includes the fanout computations, a state and I/O operations, can be defined by composing `Fanout` to some other data type constructors. The next section elaborates upon the theory behind these datatype constructors, proves some relevant properties of the `Fanout` type constructor and refers to known theoretical results to explain how to build `Action` based on simpler constructors.

4.3 Theoretical Foundations

The `Fanout` data type constructor equipped with the polymorphic versions of the functions `unit` and $\star$ defined in Section 4.2 represents a construct which is well known in the functional programming community as *monad*. The monad concept originates in *Category Theory* [90] and is defined as a triple composed by a *functor* and two *natural transformations* acting on it.

The mathematical concept of monads has been applied to computer programming by Moggi [75] and adapted to functional programming by Wadler [111, 112]. In this context, a monad is defined as a triple $(M, unit, \star)$, where:

- M is a type constructor;
- $unit$ is a function that turns a value from the type a into a computation $M\ a$ that only returns that value:
$$unit : a \longrightarrow M\ a$$
- $\star$ is an operator (also known as *bind*) that applies a function of type $a \longrightarrow M\ b$ to a computation of type $M\ a$:
$$\star : M\ a\ \times (a \longrightarrow M\ b) \longrightarrow M\ b$$

A monad satisfies the following three fundamentals laws:

1. *Left unit.* For all functions $f : a \longrightarrow M\ b$, and value a of type a:
$$unit\ a \star f = f\ a$$
2. *Right unit.* For all computations m of type $M\ a$:
$$m \star unit = m$$
3. *Associativity.* For all computations m and functions $f : a \longrightarrow M\ b$ and $g : b \longrightarrow M\ c$
$$(m \star f) \star g = m \star (\lambda a.(f\ a \star g))$$
where the notation $\lambda x.y$ denotes a (nameless) function computed on the value x that returns y.

According to the above definition of the monad, we need now to prove that the type `Fanout p` introduced in the previous section is indeed a monad, i.e., it satisfies the above monad laws.

Theorem 4.3.1. *The parametric type* `Fanout P` *defined in Definition 4.2.1 equipped with the* `unit` *function (Definition 4.2.2) and the* $\star$ *operator (Definition 4.2.3) is a monad.*

Proof. To prove that the triple $(Fanout, unit, \star)$ is a monad, we need to show that the three monad laws hold.

Left unit. By the definition of $\star$ given in Definition 4.2.3, it is easy to see that $Pass\,p \star f = f\,p,\ \forall f$.

Right unit. Again, by Definition 4.2.3, it is easy to verify that $m \star unit\, p = m$ for all possible m.

Associativity. To prove associativity, we need to verify the result of the $\star$ operation for all possible combination of pairs of functions. As we have six possible computations (Drop, Pass, Broadcast, Steer, Dispatch, Deliver), this makes 36 possible combinations. A trivial (but way too verbose to be reported here) direct check easily leads to verify that the associativity property holds. $\square$

As already mentioned, the `Fanout` monad is not sufficient to describe all the possible operations performed on packets and its effects must be combined with the effects of other well-known monads, such as the *State monad* (allowing to associate a state to each computation) and the *IO monad* (allowing for computation-driven packet forwarding). The combination [68] of such monads leads to the definition of a complete `Action` monad that will be used throughout the rest of the chapter.

The `compose` function presented in Section 4.2 operates on pairs of monadic functions. This kind of composition is generally known as *Kleisli composition*. More formally, given two monadic functions $f : a \longrightarrow Mb$ and $g : b \longrightarrow Mc$, the Kleisli composition $\circ$:

$$\circ : (a \longrightarrow Mb) \times (b \longrightarrow Mc) \longrightarrow (a \longrightarrow Mc)$$

of f and g is defined[2] as:

$$(f \circ g)\, x = (fx) \star g \tag{4.1}$$

Note that, historically, the triple $(M, unit, \star)$ used to define a monad is known as *Kleisli triple*.

The straightforward abstract consequence is to model the pure functions composing the packet processing pipeline as monadic functions that can be combined through the Kleisli composition. As a result, application programmers can rely on the functional modeling based on monads with no need to know any details about the underlying implementation. Also, this approach prevents irrecoverable errors as the strong type check of common functional languages (for example, consider the Haskell type system) guarantees the semantic correctness of compositions at type-level.

4.4 The pfq-lang Language

As discussed in the previous sections, a packet processing pipeline can be described by composing multiple functions implementing elementary operations. This leads to the definition of a Domain Specific Language (DSL) designed for packet processing (pfq-lang).

pfq-lang is based on a set of primitive monadic functions implemented in a *functional engine* that represent single stages of the processing pipeline. These primitive functions can be combined by using the Kleisli composition (see Section 4.3), which

[2]An equivalent definition may be given by using λ notation as:

$$f \circ g = \lambda x. fx \star g$$

was introduced as the `compose` function in Section 4.2 and indicated with the symbol[3] `>->` in pfq-lang.

A pfq-lang program can be either a single monadic function or a Kleisli composition of two or more operations, as:

```
c3 =  c1  >->  c2
```

As primitive operations, pfq-lang provides a rich set of functions. Also, since the functional engine is designed to be easily extensible, it allows users to add functions for their specific purposes.

Like any functional language, pfq-lang supports high-order functions (functions that take or return other functions as arguments) and currying, to convert functions that take multiple arguments into functions that take a single argument, as explained in Section 4.2.

A special function named `conditional` allows to evaluate two different computations depending on the truth value that a given predicate evaluates to. The syntax of `conditional` is:

```
conditional  predicate  c1  c2
```

where *predicate* is a function that when applied to a given packet, evaluates to a boolean value, and *c1* and *c2* are monadic functions. If the predicate evaluates to `true`, then the value of `conditional` is the expression *c1*, otherwise it is the expression *c2*. As shown in the next section, we anticipate that pfq-lang provides a rich set of predefined predicates (such as `is_udp`, `is_ip`, etc...).

For simplicity, two additional functions `when` and `unless` are also defined:

```
when   pred c =  conditional pred c  unit
unless pred c =  conditional pred  unit c
```

meaning that `when` computes *c* if *pred* is `true` or `unit` otherwise, whereas `unless` returns `unit` when *pred* evaluates to `true` or *c* otherwise.

4.4.1 Monadic functions

From a semantic point of view, monadic functions are implemented as the combination of the `Fanout` monad with other well-known monads to functional programming community (namely, the so-called *State monad* and the *IO monad*). Remember that a monadic function represents a stage of the pipeline, which takes an argument – a packet – and returns a packet along with an Action (namely, a Fanout operation, a new State and possible I/O side effects). Such functions allow performing fundamental actions on top of packets, such as filtering, forwarding, steering, classifying, marking, storing, copying, and so forth.

As discussed above, thanks to the currying mechanism, nothing prevents a monadic function from taking additional arguments, provided that they come first in the arguments list.

[3] We could not use the more typical >=> symbol as it is already used by Haskell to represent the Kleisli composition.

As a final remark, note that pfq-lang does not provide the standard $\star$ (bind) function. Instead, it does provide the Kleisli composition operator, which represents for monads what the functional composition (.) does for simple functions. While, at first sight, this may look odd, it becomes more clear when we consider that the bind function takes a monadic argument (which is a packet along with its context) and that such an argument is not available at the point where the composition is defined. Instead, the Kleisli composition allows composing a pair of monadic functions into a new monadic function whose effects are chained together. It goes without saying that the bind function (even if it is not exposed by pfq-lang) is implemented within the functional engine.

4.4.2 Non-Monadic Functions

In addition to monadic functions, pfq-lang provides a set of functions that take arbitrary arguments (curried from user-space), a packet, and return data. These functions do not specify any action on packets. Instead, they are passed as argument to high-order functions to specify their behavior. Non-monadic functions are roughly divided into the following categories: predicates, combinators, properties and comparators.

Predicates are functions that take a packet and return a boolean value. They are the fundamental building block for high-order functions, like those expressing conditional computations (e.g. conditional, `when` and `unless` functions). In addition, predicates can be combined through combinators, which are high-order functions that take one or possibly a pair of predicates and return a new predicate. pfq-lang implements combinators by overloading the common boolean operators *not* (`!`), *or* (`||`), *and* (`&&`), and *xor* (`xor`).

Similarly to predicates, properties are functions that take packets and return values, such as a computed hash, a specific field of a network header or, more in general, a value associated with the packet. Properties can be passed as arguments to comparators, which are predicates that provide a comparison between the outcome of properties and their actual arguments. Examples of comparators are: less (<), less equal (<=), equal (==), not equal (`/=`), greater (>) and greater equal (>=).

4.5 Implementation

The proposed packet processing architecture has been implemented in a Linux kernel module, and consists of a functional engine and some user-space bindings (implementing pfq-lang) for various programming languages.

The processing functions and the functional engine are implemented in kernel space as close as possible to network device drivers and the network stack of the Linux kernel.

At user-space, bindings for the Haskell language (a pure functional language, which supports monads and their manipulations), for the C++ language, and for the C language (with a slightly awkward, but functionally equivalent, syntax) are provided.

The functional engine has been implemented in the PFQ kernel module [20, 21], a high-performance monitoring framework designed for the Linux operating system.

4.5.1 The Embedded DSL

To create a programming language for a specific domain, we can either implement a parser along with the related compiler, or exploit the expressiveness of existing languages

pfq-lang has been implemented as an eDSL primarily because this choice enables a better integration with general-purpose languages already used for network applications. The eDLS implementations take advantage of the expressiveness of Haskell along with some extensions (e.g., *Generalized Algebraic Datatypes* – GADTs) and of *expression templates* for the C++ language. The lack of expressiveness of the C language has made the implementation of the eDSL more cumbersome. For example, the Kleisli composition is implemented as a traditional function instead of using a custom infix operator.

At the time of writing, pfq-lang implements about a hundred functions, and more are expected in the future. As previously mentioned, such functions include protocol filters, steering functions, conditional functions, or forwarding functions either to sockets, devices or the kernel stack. Due to space limitation, only a few functions are shown in the chapter. The complete reference of the available functions can be accessed at the pfq-lang wiki page [11].

As an example of pfq-lang expression, a simple function that filters IP packets and dispatches them to a group of endpoints (e.g., sockets) using a steering algorithm is described as:

```
composition = ip >-> steer_ip
```

where `ip` is a filter that drops all the packets but IP ones, and `steer_ip` is a function that performs a symmetric hash with IP source and destination. Such a composition can be embedded in a Haskell program, associated with a given group of endpoints (identified by the *gid* parameter) and passed to the functional engine in kernel-space (where it is validated and executed on a per-packet basis) with the following command:

```
Q.groupFunction q gid composition
```

where `Q` is the namespace used for the PFQ and PFQ-Lang symbols, and `q` is an instance of a PFQ socket type (used as a *handle* to the functional engine).

The C++ version of eDSL uses a slightly modified syntax:

```
auto q = net::pfq();
...
auto composition = ip >> steer_ip;
q.groupFunction(gid, composition);
```

where, again, `q` is an object representing a PFQ socket (used to access the functional engine).

The main difference between the pfq-lang syntax and the C++ eDSL syntax is in the fact that the Kleisli composition in the C++ eDSL is represented by the $>>$ operator (other differences are the extra parenthesis in the function used to inject the computation into the functional engine, but these are not strictly part of the eDSL). Notice that from now on we will only report examples written in the native pfq-lang syntax (which, incidentally, coincides to that of Haskell).

pfq-lang implements filters for the most important protocols; to name a few, `ip`, `udp`, `tcp`, `icmp`, `ip6`, `icmp6`, `rtp` (heuristic) and so forth. In addition, each filter is complemented with a predicate, whose conventional name begins with `is_` or `has_`.

Conditional functions allow to change the behavior of the computation, depending on a property of the processed packet, as in the following example:

```
composition = ip >->
    when is_tcp
        forward "eth1" >-> steer_flow
```

The function drops all non-IP packets, forwards a copy of TCP packets to `eth1`, and then dispatches packets to the group of registered PFQ sockets in steering mode. It is worth noticing that since `steer_flow` works with both TCP and UDP packets, ICMP packets are implicitly dropped by the steering function.

As another example, the following code marks UDP packets with the number 42 (or returns them to the kernel network stack if non-UDP packets), and then dispatches them to PFQ sockets preserving the flow integrity:

```
composition =
    conditional is_udp
        mark 42
        kernel >-> steer_flow
```

The `mark` function is used to mark packets with a value that can be read by following functions through the `has_mark` predicate as well as by user-space applications when packets are received by sockets. This marking mechanism can be used for packet classification.

Finally, it is worth noticing that the whole *Berkeley Packet Filter* machinery can be included into a single monadic function and used in functional compositions.

4.5.2 The Functional Engine

At the bottom of the system, PFQ [21] retrieves packets from NICs *data sources*. Such packets are then fed into the functional processing pipeline that provides the kernel stage of processing (filtering, classification, steering toward upstream applications, etc.). At the end of the pipeline, PFQ sockets, NICs, or the kernel networking stack can be used as endpoints to feed applications running in user-space, forward packets, or return them to the kernel.

User-space applications must register to *socket groups* to receive packets. Socket groups allow for multiple multi-threaded (or multi-processes) applications to concurrently receive packets from the same set of NICs. As such, each group is served by a specific processing pipeline (an instance of functional composition).

The engine evaluates the functional composition specified for an endpoint (or a group of endpoints) on a per-packet basis. The number of instances of the functional engine available in a system depends on the number of kernel contexts enabled for capturing packets, that in the Linux kernel corresponds to the number of NAPI threads. A fine-grained approach to interrupt affinity allows selecting the cores where the functional engines are instantiated.

The functional composition is represented by an abstract syntax tree (AST) of functions and arguments, and it is generated in user-space by pfq-lang expressions. The AST is then represented regarding a collection of meta information to be transferred to the functional engine (through a specific PFQ socket option). Such information includes the actual parameters specified in user-space upon their conversion to a memory layout compliant to the C language.

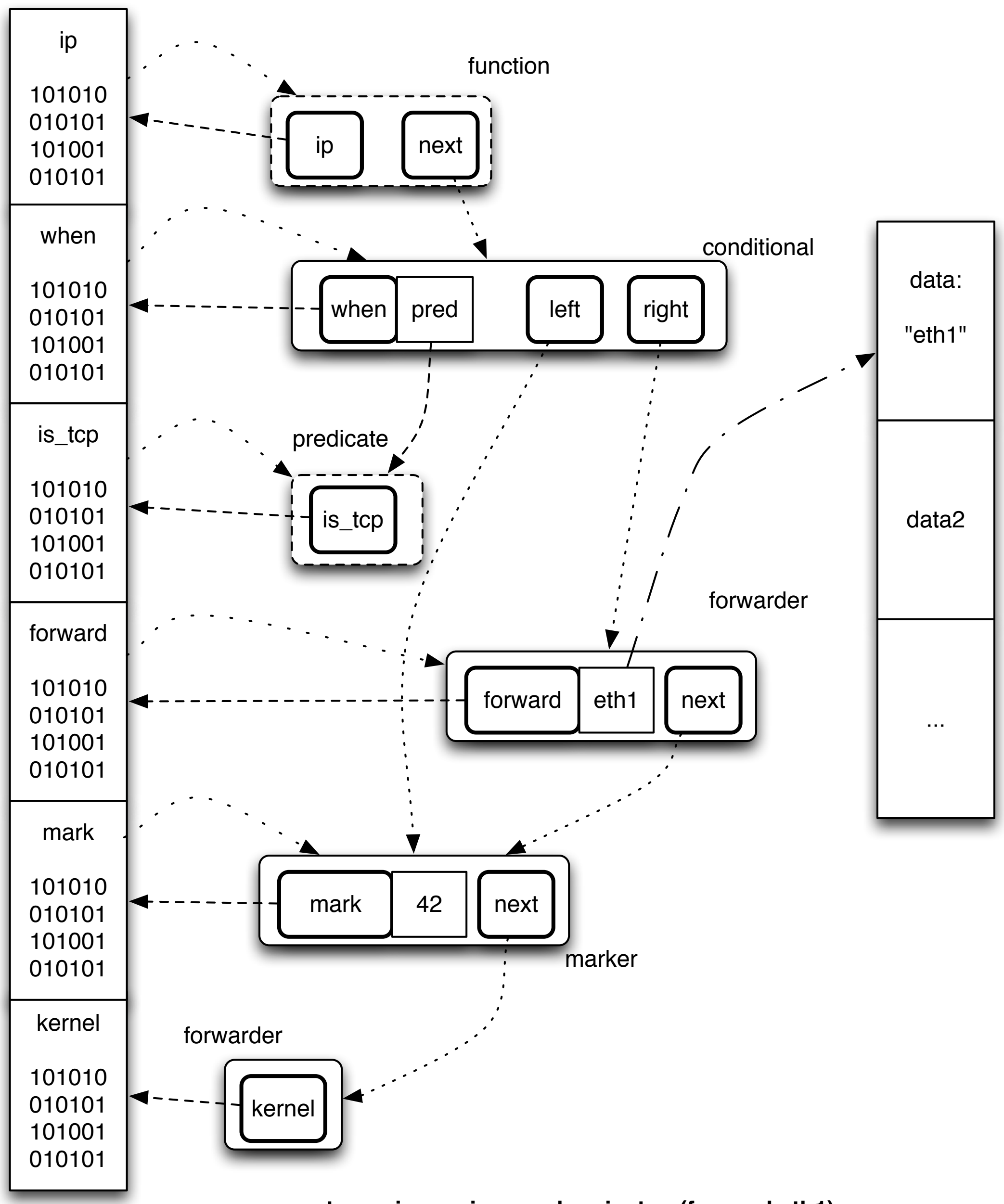
The eDSL enforces the type level correctness at compile time. Also, for security reasons, the functional composition is sanity checked at kernel-space to avoid possible kernel panics. In particular, the signature of the various functions along with their argument types are verified, and the presence of loops in the tree is avoided. Subsequently, the descriptors of the various functions undergo a process of transformation through which the abstract tree is converted into a hybrid data structure, equipped with data and pointers to executable functions (see Figure 4.2). The addresses of various functions are resolved by a run-time linker that uses dynamic symbol-tables. Such tables are populated when either the PFQ kernel module or external modules are loaded. Notice that external modules can add (and remove) on-the-fly new functions to the system (the removal is allowed only when no functional engine is running).

The curryfied arguments of such functions are stored in a functional node. For performance reasons, depending on their size (as compared to that of a pointer) either their value or a pointer to a separate chunk of memory is stored in the node. Once the AST is converted into the executable program, the functional composition is transactionally enabled for a specified group of endpoints with a single atomic operation. This enables computations that can be dynamically updated in real-time, for example in response to network events.

The primitive operations implemented in the functional engine are divided into different symbol tables based on their prototypes. Monadic functions take as argument a socket buffer (`sk_buff` for the Linux kernel) and return the `sk_buff` enriched with a context (representing action, state, and logs for I/O). Predicates can receive some optional arguments and a `sk_buff` and return a boolean value. Properties return a value associated with the packet. More in general, functions can access the `sk_buff`, some arguments specified by the functional composition in user-space (which are curryfied), possibly additional functions passed as arguments and a state associated with the packet which is stored in the control buffer of the `sk_buff`. In particular, the control buffer hosts some meta information that is used to implement the `Fanout`, the `State` and the `IO` monads used to perform actions on the packet.

Note that the `Action` monad includes the effects of a *State monad*, implementing two different kinds of states: volatile and persistent. The volatile one is used to store information related to the current computation and is passed along the chain of functions. Such a state is available to all the functions and the user-space applications when the packet and its meta-data are passed to the sockets. The volatile state has a scope limited to the composition, it is thread-safe and represents a mechanism for classification that can be triggered by in-kernel functions.

As an example, we report the in-kernel function that is executed once the monadic function `l3_proto` is evaluated by the functional engine. The function implements a filter for the layer 3 protocol specified as argument (e.g., `l3_proto 0x800` to filter for IP packets).



**tcp_mirror = ip >-> when is_tcp (forward eth1) >->
mark 42 >-> kernel**

Figure 4.2: *Memory layout of the executable program*

```
Action_sk_buff
filter_l3_proto(arguments_t args,
                struct sk_buff *skb)
{
    const u16 type = get_data(u16, args);

    if (eth_hdr(skb)->h_proto ==
        __constant_htons(type))
            return pass(skb);

    return drop(skb);
}
```

The function uses the standard methodology of parsing packets used in the Linux kernel. The return value, instead, specifies a computation by using the functions defined in the `Fanout` monad (i.e. `pass` and `drop`). The type `arguments_t` is an additional argument used to mimicking currying at the kernel level. The function may access to the curryfied arguments using the appropriate macro, e.g., `get_data`.

4.6 Performance Evaluation

The performance of the pfq-lang implementation presented in Section 4.5 has been evaluated through an extensive set of experiments. All the experiments have been performed on a PC equipped with an Intel 82599 10G NIC having 128 hardware queues and based on a 6-core Intel Xeon X5650 running at 2.67GHz. This means that without hyper-threading, the maximum number of parallel instances of the functional engine (processing pipelines) that can be executed simultaneously is 6. Since the more recent release of PFQ does not benefit from hyper-threading, processing pipelines have been tested by varying the number of hardware queues (and therefore the NAPI contexts) from 1 to 6.

The performance has been measured by feeding the PC running PFQ, the functional engine, and the pfq-lang coded applications, with packets generated at increasing rates, and measuring the rate of processed packets. The packet length is set to 64 bytes (worst-case).

In the first experiment, the performance of an empty processing pipeline has been measured to have some baseline performance to be used as a reference. This experiment measures the PFQ performance in capturing packets and the minimum overhead introduced by the functional engine. Figure 4.3 shows the results. As can be noticed, on the hardware available to us, PFQ can capture packets at line rate from a 10Gb Ethernet card by using 4 CPU cores with vanilla drivers (it is worth remembering that this corresponds to around 14.8 Mpps). When 3 CPU cores are used, PFQ can capture up to 14 Mpps. 2 CPU cores can capture up to 10 Mpps, while a single CPU core can reach about 5.5 Mpps.

After measuring the performance of PFQ and an empty computation, the performance of the simplest possible function (a single `unit` action) has been evaluated. The results are reported in Figure 4.4 and show that introducing a simple action in the pipeline slightly reduces the performance achieved when using 2 CPU cores or 3 CPU cores, while the performance of a single standalone pipeline does not seem to be significantly affected. The functional engine scales (concerning the number of instances)

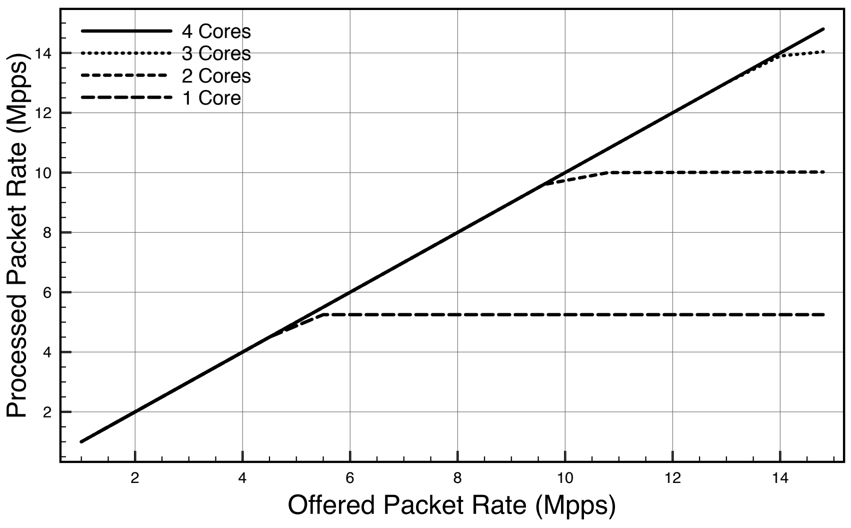


Figure 4.3: *Empty processing pipeline*

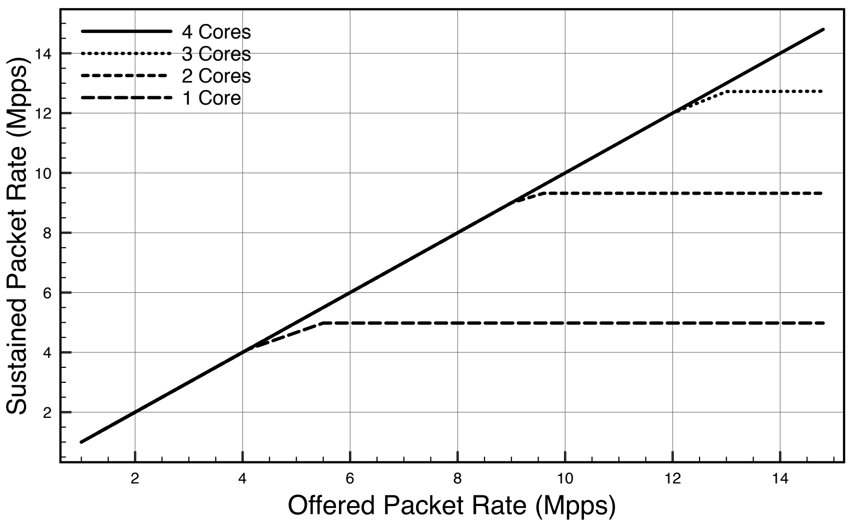


Figure 4.4: *Processing pipeline consisting of a single unit function*

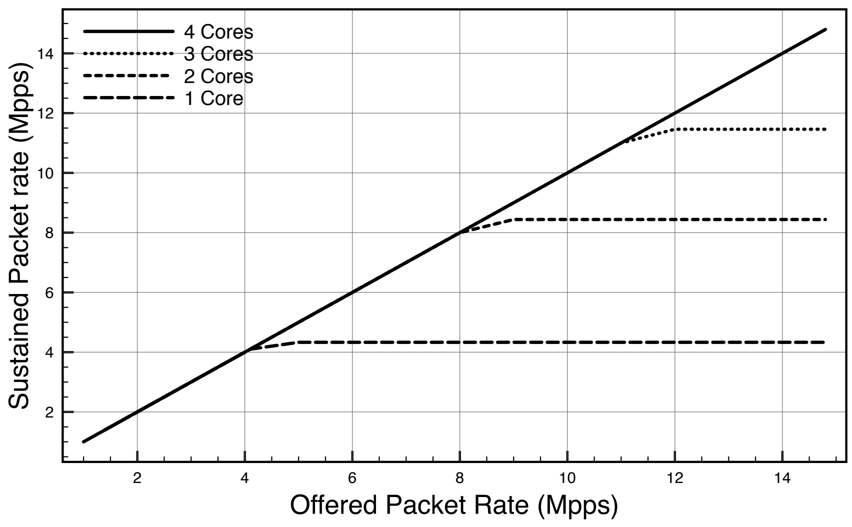


Figure 4.5: *Functional computation:* `(when is_tcp (mark 42)) >-> steer_ip`

slightly worse than in the first case, but still scales pretty well.

Then, the performance of the functional engine when executing a more complex computation (including a conditional action and the delivery of packets to user-space) has been evaluated. The function used in this case is

```
(when is_tcp (mark 42)) >-> steer_ip
```

which marks the TCP packets with value 42 and then delivers all the packets to user-space by using a steering function. The packet delivery is obtained by calculating a symmetric hash on the IP source and destination addresses, thus forcing packets generated between the same pair of hosts to be always delivered to the same endpoint(s). Figure 4.5 shows that the engine is again able to process (and deliver to user-space) packets at line rate when using 4 CPU cores. As expected, the performance achieved when using 1, 2, or 3 CPU cores are slightly worse than in the previous cases, indicating that the increased complexity of the computation can affect the number of packets per second that the engine can process.

Figure 4.6 shows the performance of computation which performs forwarding to a network device, and then drops all the packets (hiding them to the kernel). This experiment measures the overhead introduced by the I/O monad (the cost of sending a packet), showing that the system can reach a forwarding rate of around 12 Mpps with the hardware available to us.

Notice that, so far, all the tested computations consumed a very small amount of execution time. In the last experiment, the performance achieved when using a CPU-intensive computation (computing the CRC16 on all of the bytes of each packet) has been evaluated. The results are reported in Figure 4.7 and show that the system still scales linearly with the number of cores and reaches around 13 Mpps processing rate on the hardware available to us.

As a final remark, it is worth noticing that in all the experiments the performance lines become flat after the peak throughput is reached (in other words, when the max-

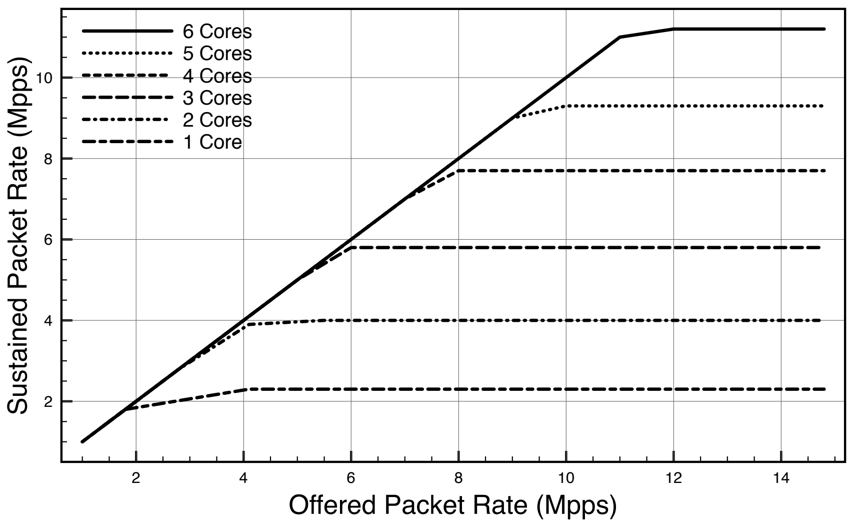


Figure 4.6: *Functional computation:* `forward "eth2" >-> drop`

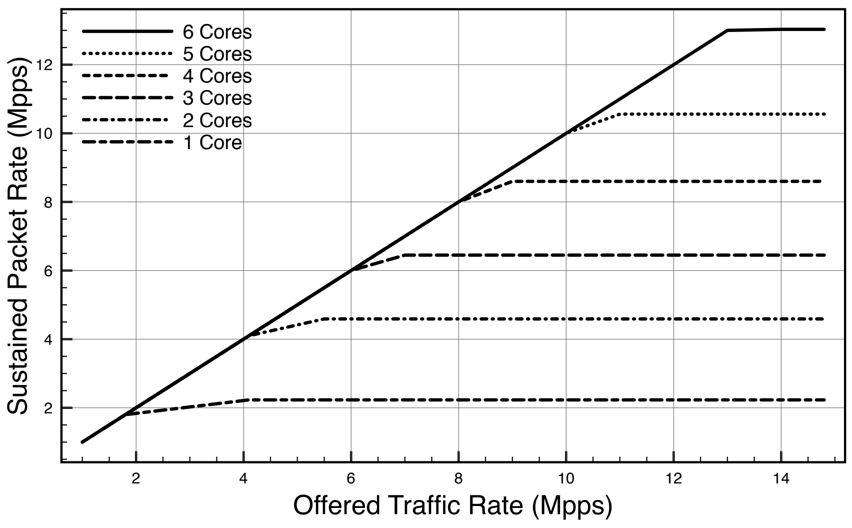


Figure 4.7: *Heavier functional computation: CRC evaluation*

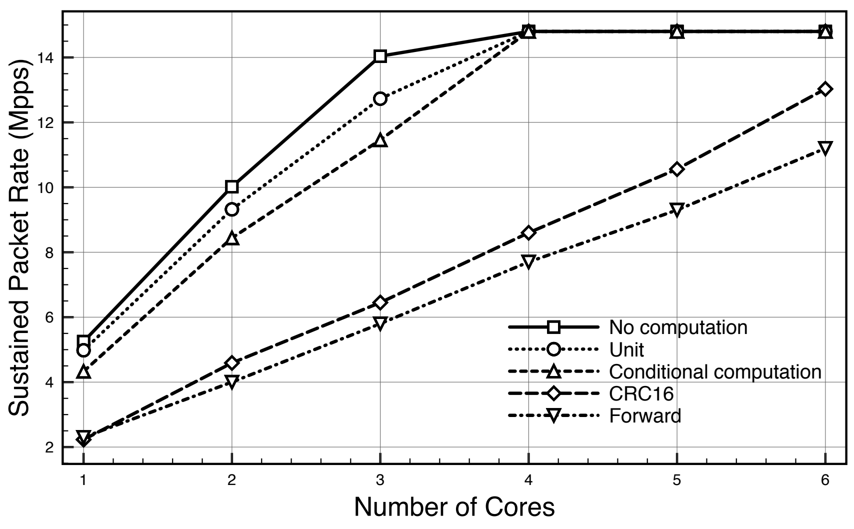


Figure 4.8: *Scalability Overview*

imum “Processed Packet Rate” is reached, increasing the “Offered Traffic Rate” does not decrease the performance). This shows that PFQ and pfq-lang work well even in overload conditions. Figure 4.8 shows the peak throughput measured in the various experiments as a function of the number of used CPU cores and tells us that the pfq-lang/PFQ combination can properly scale with the number of cores.

4.7 Use Cases

This section provides some examples of possible network applications that can be implemented by programming the functional engine. Such examples range from packet forwarding, port mirroring, a load balancer switch, stateless and stateful firewalls, and two simple early-stage processing applications for network monitoring.

4.7.1 Port mirroring

Since it is possible to use a NIC as one of the endpoints, packets can be quickly forwarded to other machines by merely directing them to the proper endpoint. This strategy allows to efficiently implement the data plane of a switching node through a set of pure functions.

The combined use of the *IO monad* [52] and a suitable garbage collector allows the actual forwarding operations to take place at the end of the processing pipeline, even if some intermediate stages might return a different packet. This way, the purely functional “lazy” nature of the pipeline stages is preserved. Furthermore, it turns out that lazy forwarding leads to performance optimization as, the a posteriori knowledge of the number of forwarding operations requested for a packet, always saves one shallow copy of the packet itself.

A mirroring port can then be simply instantiated through the following expression:

```
forward "eth1" >-> kernel
```

where the ports to mirror are specified when the group of sockets (for which the functional composition is specified) is bound to a list of devices.

4.7.2 Load Balancer

The use of randomized operations easily allows the implementation of a network *load balancer* when endpoints are network devices. To enable this, the application needs to associate NICs to special endpoints (egress sockets) to be used for the fanout. Hence, a very simple flow-based load balancer can be implemented by a single steering function:

```
steer_flow
```

alternatively, by any other steering or dispatching functions available from the language.

4.7.3 Stateless Firewall

A pipeline of purely functional filters can be used to implement a firewall. Filters can drop packets, or set them to be re-injected in the protocol stack (so that they are received by the system).

For instance, functional filters can be configured to:

- block the packets explicitly recognized by the pipeline. In this case, all the received packets are set by the first stage of the pipeline to be re-injected in the protocol stack, and some of the following stages can explicitly drop the packets to be blocked by the firewall;

  ```
  kernel >-> (when has_port 22 &&
              !(address "131.114.0.0" 16) drop)
  ```

- block all the packets except for the ones recognized by the pipeline. In this case, by default, packets are not marked in any way (i.e., by the first stage of the pipeline), but are explicitly marked as “re-inject in the protocol stack” when recognized by some filter.

```
        when has_port 80 kernel
```

Notice that in this case, the firewall is stateless, because packets are received or dropped upon their properties, and not by previous packets.

4.7.4 Monitoring

pfq-lang has also been designed for network monitoring. By leveraging the features/performance of PFQ in capturing packets from multiple NICs (or from multiple hardware queues), pfq-lang enables the creation of efficient early stages computations for more complex monitoring applications.

pfq-lang includes a set of functions specifically designed for packet filtering and steering, covering the most used protocols and heuristics. For instance, an early stage computation suitable for an application that estimates the network parameters of RTP flows (i.e. packet loss, end-to-end delay and jitter) can be instantiated as:

```
conditional is_rtp (class 0 >-> steer_rtp)
                   class 1
```

The heuristic `is_rtp` is used to detect VoIP flows even in the absence of RTCP traffic, and `steer_rtp` is the steering function used to dispatch RTP/RTCP packets across the endpoints in use in the class 0 (a subset of endpoints of the group for which the composition is specified). Traffic not detected as RTP/RTCP is sent to a subset group of endpoints (class 1) designed to receive packets for other purposes (e.g., debugging).

4.7.5 Legacy applications

PFQ comes with a full-featured pcap library, enabling legacy applications to benefit from its features. The pcap interface is mapped over the rich PFQ APIs, and when no match is possible – that is when no pcap function is available to wrap PFQ APIs – other mechanisms, such as environment variables are adopted. As a result, even legacy applications can benefit from pfq-lang to create early stages programs for network applications.

Among the predefined steering functions, a very useful one is `steer_net`. Such a function takes a string as network address (IP) along with a prefix (as the second argument) to identify a network. The last parameter is a second prefix used to further subnet such a network and to steer the related packets to different endpoints.

```
        steer_net "192.168.0.0" 16 24
```

In the reported example, the traffic that belongs to the 192.168.0.0/16 network is selected, split into 256 C-classes, and finally steered to the endpoints.

The PFQ pcap interface is extended with an environment variable that allows launching applications specifying for them the group id of their sockets. Therefore once the above program is specified for a group through a control socket, say group 42, it suffices to launch multiple sessions of the legacy applications and let them see the traffic properly split.

A typical example is that of multiple *snort* sessions:

```
# PFQ_GROUP=42  snort -c /etc/snort.conf \
    -l /var/log/snort1/
# PFQ_GROUP=42  snort -c /etc/snort.conf \
    -l /var/log/snort2/
# PFQ_GROUP=42  snort -c /etc/snort.conf \
    -l /var/log/snort3/
```

4.8 Stream processing

The rise of Software Defined Networks (SDN) and Network Function Virtualization (NFV) as the winning trends for the design and development of network functions and services has naturally switched the focus of the research community towards network virtualization and softwarization. Both paradigms propose a dramatic change in the way network operations and services are conceived and push *programmability* and *reconfigurability* as crucial keywords of a new generation of network devices.

This section presents Enif-lang (Enhanced Network processIng Functional Language), a natural, expressive and robust programming language specifically tailored to network traffic processing for multi-core PCs running Linux OS. Originated from the experience learned with PFQ-Lang [19, 23], the language has evolved quite a lot lately and the current prototype implements a reference model that *enables and automates state persistent in concurrent programming*. Like any functional language, Enif-lang supports high—order functions (functions that take or return other functions as arguments) and currying, that turns functions that takes multiple arguments into functions that take a single argument. Moreover, the language includes conditional functions and predicates to implement a basic code control flow.

Since Enif-lang is used to describe and specify packet processing pipelines, it plays a similar role to that of the lower level P4 [24] language and to the imperative language Pyretic [76] (that belongs to the Frenetic [8] family of network programming languages) in describing the data plane logic of an SDN network or to that of Streamline [37] to configure I/O paths to applications through the operating system. Also, VPP [10] and eBPF [2] recently proposed alternative approaches for packet processing and data plane programming.

4.9 Enif-lang at a glance

Enif-lang is a functional language entirely implemented as a declarative Domain Specific Language (DSL) on top of the Haskell Language and it is designed to ease the implementation of network applications by leveraging a strong type-safe system and the functional composition typical of functional programming languages.

Figure 4.9 shows the full Enif-lang abstract processing model implemented in the runtime. Packets, retrieved at physical network interfaces, traverse a splitting layer that provides per-flow consistency and are injected into the processing engines (the λ_i blocks in the picture). At this stage, the pipelines of computation are executed according to the formal description provided by Enif-lang and packets are finally forwarded to the selected *endpoints*, i.e., to network cards, application threads, the OS kernel, etc..

In Enif-lang, pipelines are formally expressed as the composition of effectful functions, called *actions*, that perform network operations on top of packets. In analogy

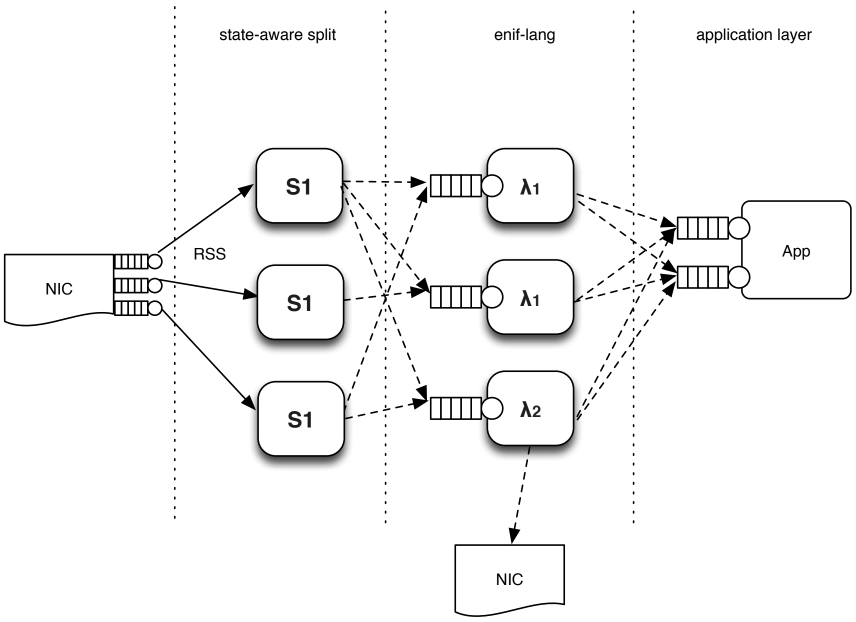


Figure 4.9: *The Enif-lang abstract processing model*

with PFQ-Lang [23], Enif-lang actions are formally modeled around the concept of *monad*, a structure borrowed from the Category Theory and widely used in other languages, such as Haskell, Scala etc. In short, monads provide the theoretical support for composing effectful functions (such as those performing I/O, those that handle a state associated with a packet or flow, and so on) while maintaining the language functionally pure. Enif-lang actions include the most common packet processing primitives for packet forwarding, filtering, steering, logging and statistics retrieval.

At the time of writing, an experimental implementation of the reference architecture runs on top of the PFQ framework [19] for the Linux operating system. In particular, the Enif-lang run-time takes advantage of the GHC Haskell compiler to produce intermediate representations of Enif-lang programs that are loaded as shared libraries and executed in user-space, on a per-packet basis.

The following sections describe the basic grammar and syntax of the language along with simple snippets for its practical usage.

4.9.1 Functions overview

Enif-lang is equipped with a set of built-in primitives as well as a complete library of functions to describe generic processing pipelines. In the following, such functions are roughly divided into different categories according to their purposes.

Predicates. Predicates are pure functions that take an arbitrary number of arguments (possibly none) plus the current packet and return a Boolean value. Such functions are either used within the `if-then-else` statement or passed as an argument to high-order-functions to specialize their behavior.

The language implements a set of primitive predicates (that cannot be directly implemented in Enif-lang) that compose together to implement more complex ones.

The default library includes predicates for the most common protocols. As a convention, the names of such functions are prefixed by `is_` or `has_`, when meaningful.

Examples are `is_tcp, is_udp, is_icmp, is_rtp, is_sip, is_gtp` or `has_port 80, has_addr "192.168.0.0/24"`.

Combinators. Enif-lang provides a set of combinators, that is functions designed to combine predicates together. In particular, the composition of predicates is enabled by the logical `or`, `and`, `xor` and `not` functions.

Properties. Properties are functions designed to return a value associated with a packet. Typical examples are hash functions computed over a portion of a packet, header field extractors, state retrievals, etc. The Enif-lang library is equipped with a wide gamma of property functions for the most common protocols (IP, TCP, UDP, ICMP), as well as with generic functions that extract the field value of arbitrary protocols (by specifying the offset and the size of the field). For example, properties of the IP header are: `ip_ttl`, `ip_tot_len`, `ip_id`, `ip_frag`, `ip_ttl`.

Comparators. Properties are meaningful only when used with comparators, namely functions that perform a comparison between a given property and a specified value. In addition to the standard operators <, <=, >, >=, == and `/=`, the library offers `any_bit` and `all_bit` functions to check whether some (or all) bits of a given mask are set. As an example, the expression `any_bit ip_tos 0x3f` is a valid Enif-lang predicate that tests whether any of the DSCP bits are set in the packet.

Filters. Filters are effectful functions that break the pipeline processing when the packet does not match a given condition. The Enif-lang library is equipped with a wide range of filters for the most common protocols. In a nutshell, a filter is a very simple monadic function, whose output action can be either Pass or Drop. Examples of common filters are `ip`, `tcp`, `udp`, `port`, `src_port`, `dst_port`, `addr`, `src_addr`, `dst_addr`, etc.

Monadic functions. Monadic functions take an arbitrary number of arguments and a packet and return a packet with an associated action. Currently, the available actions are Pass, Drop (used by filters), Broadcast, Dispatch, Steer and DoubleSteer (used by steering functions).

Common monadic functions are `when` and `unless`, used in conditional statements as well as the family of steering functions, such as `steer_flow`, `steer_p2p`, `steer_link`, etc., used to balance the traffic among multiple endpoints with different flow consistency guarantees.

4.10 Processing Pipelines

Actions can be composed together by means of the `do` notation or through the Kleisli operator >=>, and follow similar rules of composition set forth in [23].

The overall expressiveness of Enif-lang allows building even complex pipelines through a very concise grammar. As a first example, the following simple program:

```
udpCounter = counter 0
enif_main :: Packet -> Action ()
enif_main pkt =
    when (is_udp pkt) $ modifyCounter udpCounter (+1)
```

which makes use of the predicate `is_udp` to count UDP packets. The entry point `enif_main` is the principal function invoked by the Enif runtime, to which the packets captured from the NICs are passed.

The next snippet of code, instead, provides a more complex example in which the processing pipeline takes advantage of a per-computation state – that is a state passed along the processing stage.

```
http = dst_port 80
pass_to_kernel =
    when (has_state http_traffic)
         kernel
mirror_to_port =
    when (has_state other_traffic)
         (forward "eth2")
process = if (not is_tcp)
            then drop
            else do pass_to_kernel
                    mirror_to_port)
http_traffic = 1
other_traffic = 2

enif_main :: Packet -> Action ()
enif_main pkt =  do
  if (http pkt)
        then put_state http_traffic
        else put_state other_traffic
  process
```

From the language point of view, functions like `put_state`, `get_state` and the predicate `has_state` act as an implicit extra parameter for all of the pipeline functions. However, the usage of the per-packet state is somewhat limited as it vanishes after the packet computation expires. As such, it cannot be used in processing pipelines that require the storage of stateful information across different packets. The solution of this issue is represented by per-flow persistent states and is described in the next section.

4.10.1 Stateful Pipelines

Generally speaking, stateful operations in parallel architectures require an effective management of potential data sharing across multiple threads of execution to avoid race conditions. In the case of stateful processing pipelines, this would be the case of different packets belonging to the same flow but processed by different cores concurrently.

For network applications, it turns out that, most of the time, it is possible to tackle the problem in a general and effective way by restricting the association of stateful information to packet flows only. As a consequence, in many practical applications, it is possible to partition *a-priori* the traffic and let all the computation process flows of packets in parallel and total isolation.

The central concept is here represented by a suitable definition of packet flow. Packet flows are defined through flow-keys (e.g., IP addresses, canonical 5-tuples, and so on). In the Enif-lang context, a generic flow-key consists of the concatenation of an arbitrary number of packet header fields. Different pipelines may operate different types of

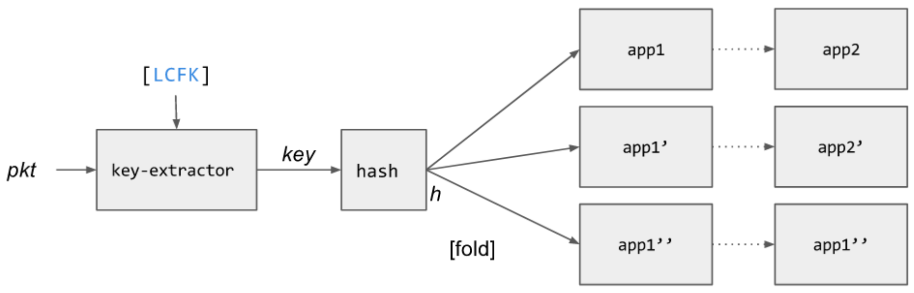


Figure 4.10: *The Enif-lang distribution model*

flows, all of them specified by their flow-keys. The bitwise intersection of all such keys represents the common flow-key that can be used upon hashing at the splitting stage (see Figure 4.9) to distribute packets to functional engines. Once traffic is split, the current abstraction guarantees that packets of the flow are processed in the Enif-lang stage sequentially, on a single core and in total order.

The figure 4.10 further describes this concept in a block diagram. The key-extractor is configured with the LCFK (largest-common-flow-key), that is the bitwise representation of the common mask in use by all applications (app1 and app2 in the figure) to identify the state associated with a certain representation of the flow.

Such a block, using the bitmask, extracts the essential part of the packet headers necessary to compute a hash (the algorithm used is not relevant) that later is used in the fold operation to split the traffic across an arbitrary number of cores. Each core is in charge to run an independent replica of the applications, sequentially.

This mechanism automatically ensures the flow consistency for all the packets and their related states and prevents from data-sharing among cores. Notice that multiple Enif-lang programs can instead run in parallel (on different cores) thanks to the immutability of packets.

However, it is worth noticing that not all configurations can be parallelized as a common flow-key might not exist, or in other words be empty. This particular case can be conveniently handled by partitioning the applications in clusters of common subkeys and by introducing shallow copies of packets to feed each of them (figure 4.11). In this example, app1 and app2 sport a common flow mask, whereas app3 is based on an orthogonal flow-concept, hence, with an independent flow-mask and set of cores where it runs.

From the language point of view, Enif-lang provides a persistent per-flow state that is automatically handled by the underlying abstract processing model. A per-flow persistent state is a state shared among all packets that belong to a certain flow. Such a state information is stored in associative flow-maps, indexed by their own flow-keys.

The Enif-lang library provides several functions for the per-flow state management. In general, all such functions take a flow map object that contains a table identifier and a flow-key definition. Furthermore, the language offers a set of predefined keys as well as

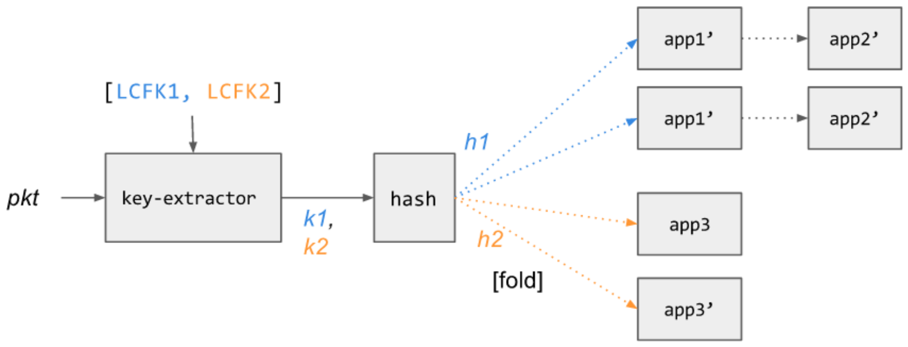


Figure 4.11: *The Enif-lang distribution model: two clusters*

utilities for building custom keys through the concatenation of arbitrary header fields. In particular, the function `set_fstate` is used to set the value of the state associated with the flow of a packet. Instead, the function `get_fstate` retrieves the value of the state associated with the flow the packet belongs to. Other additional functions, such as `incr_fstate`, `decr_fstate`, `add_fstate`, etc. are used to update the state information. It is worth noticing that Enif-lang does not limit the number of state tables whose maximum number is instead enforced by the underlying implementation.

4.11 Use-cases

This section presents the use of Enif-lang in two practical applications. The first application is an example of stateless processing and consists of a simple load-balancer that forwards packets to a cluster of network devices or local applications running Deep Packet Inspection. The second example, instead, provides the Enif-lang implementation of the simple stateful firewall based on the *port knocking* scheme.

4.11.1 Stateless processing

DPI applications typically take advantage of DNS packets to build classification trees and improve application recognition. As such, the following load-balancer broadcasts DNS packets to all DPI workers and randomly spreads the remaining packets according to `steer_p2p` steering function that preserves layer 3 symmetric flow consistency.

```
is_dns = has_port 53
enif_main pkt = if(is_dns pkt)
       then broadcast
       else steer_p2p
```

4.11.2 Stateful processing

The simple port knocking application is presented. The problem, already described in [16], is to create a simple firewall that permits certain flows to pass only if a known sequence of TCP packets hit predefined ports (port numbers 5123, 6234, 7345 and 8456 in the example).

The example uses a couple of tables. The first table lists the authorized flow and is implemented as an associative map based on the classic 5-tuple key (predefined as TUPLE_5). The second table, instead, is based on the 3-tuple keys (IP_SRC, IP_DST and PROTOCOL, defined as TUPLE_3) and implements the state machine for tracking the knocking sequence. Any time the expected destination port is found, the state is updated through the function `next_if` until it reaches the state value 4 which opens the firewall and the corresponding flow is authorized in the `auth_flow` table.

```
auth_flow   = flow_map 0 TUPLE_5
knock_table = flow_map 1 TUPLE_3
next_if pred =
    if pred
    then incr_fstate knock_table
    else set_fstate knock_table

enif_main :: Packet -> Action IO ()
enif_main pkt  =
  if (get_fstate auth_flow)
  then kernel
  else case_of (get_fstate knock_table) $ with
   [ 0 ~> next_if (dst_port. == 5123) >-> drop
   , 1 ~> next_if (dst_port. == 6234) >-> drop
   , 2 ~> next_if (dst_port. == 7345) >-> drop
   , 3 ~> next_if (dst_port. == 8456) >-> drop
   , 4 ~> do
           next_if (dst_port. == 22)
           when (set_fstate knock_table. == 0)
           (set_fstate auth_flow 1 >-> kernel)
    ]
```

A final example is that a simple flow tracker, whose source code is herein reported. The program is a simple packet counter, on a per-flow basis.

```
flowMap :: HashTable FlowKey5 Int
flowMap = newHashTable 1000000

enif_main :: Packet -> Action IO ()
enif_main pkt = do
     let key = mkFlowKey5 pkt
          e <- lookup flowMap key
         case e of
            Nothing -> insert flowMap key 1
            Just x  -> insert flowMap key (x+1)
```

Here a flowMap is declared as a `HashTable` with the `FlowKey5` tuple as key. Such a tuple, along with the utility function `mkFlowKey5` is implemented in a common Enif library.

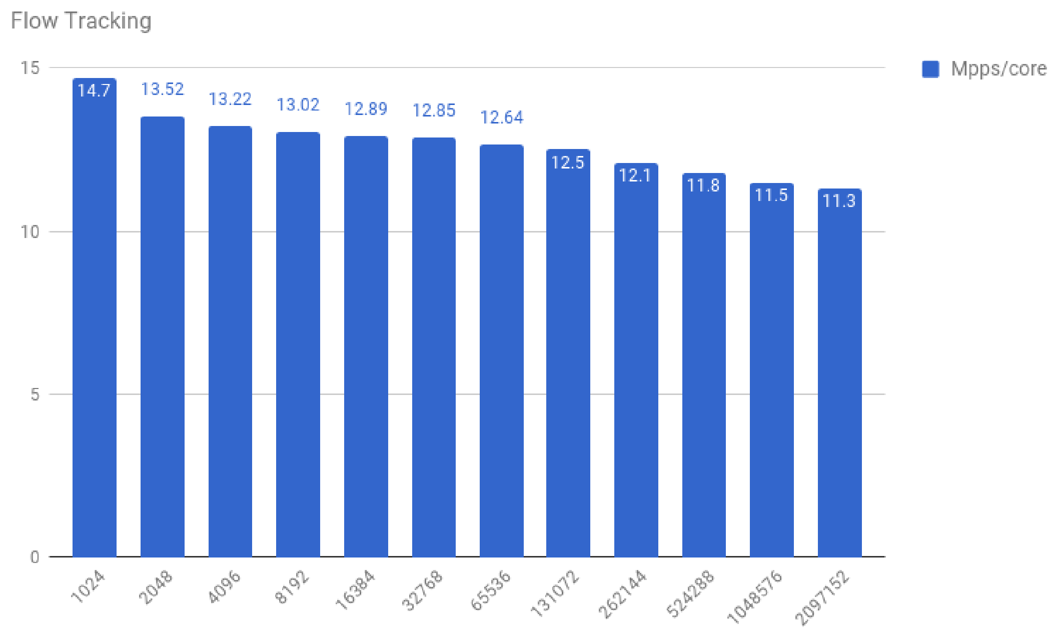


Figure 4.12: *The Enif-lang flow tracker performance*

The enif_main program is passed packets, which are parsed to extract the key necessary to increment the corresponding counter or to create an entry and set the related counter to 1, otherwise.

For completeness, the following snippet presents the implementation of the mkFlowKey5 utility function, implemented in the Language.Enif.Flow module.

```
mkFlowKey5 :: Packet -> FlowKey5
mkFlowKey5 pkt = FlowKey5 byteSwap32 (ipSaddr pkt)
                  byteSwap32 (ipDaddr pkt)
                  byteSwap16 (srcPort pkt)
                  byteSwap16 (dstPort pkt)
                  protocol pkt
```

The figure 4.12 reports on the performance achievable on a single core of our 3Ghz Xeon architecture running the flow tracker example. The test is executed generating a variable number of random flows, of 64-byte length packets. It is worth noticing that a single core can track the very worst case of 14.8Mpps when the number of flows is up to 1024, and the performance degrades, but not excessively until it reaches 11 Mpps when the number of streams is around 2 Million.

CHAPTER 5

Applications

This chapter presents use-cases and applications of the network programming elements shown in the previous chapters.

In particular, the first use-case presented takes advantage of the PFQ framework (presented in chapter 2) which is used, along with some additional performance optimization techniques, to speed up a software-based OpenFlow switch.

The second case of use, instead, uses PFQ as a measuring instrument to demonstrate the validity of a fair-bandwidth allocation algorithm among a set of TCP sources, on a set of 10G links.

Finally, we conclude with a set of distributed probes that takes advantage of the proprieties of LogLog counters and reversible sketch, to implement a system for network anomaly detection.

5.1 SDN and OpenFlow

5.1.1 Introduction

The need to quickly adapt network functions to new requirements and emerging applications has fueled recent research in programming models and abstractions for programmable network devices' data planes [16,25,74,96,99]. In this field, an abstraction based on a pipeline of Match-Action tables (MATs) emerged as an effective way to model the forwarding behavior of a network function [24].

Software switches are a natural fast prototyping tool for performing research on MAT abstractions. They typically implement a general forwarding plane that can be easily modified and used to functionally test new features before embracing longer development efforts. Nonetheless, the process of designing or modifying an abstraction very often requires a fine-tuning of a flexibility versus performance trade off [86].

Therefore, functional testing has to be performed together with performance testing.

Unfortunately, wide-spread software switch implementations, such as OpenVSwitch (OVS), achieve performance by performing some optimizations which are specific to the implemented forwarding abstraction [88]. For instance, OVS applies a complex caching strategy tailored to the implemented *stateless* OpenFlow-like MAT abstraction. Such strategies, while being effective in providing the required performance, complicate the process of introducing a new programming model that differs from the implemented one [60]. On the other side, modular high-performance switches implementations, such as mSwitch [56], or high-performance packet I/O frameworks, such as DPDK [42] and netmap [93], provide a better ground to develop a new forwarding abstraction. Still, they usually require a more substantial development effort, since they do not necessarily implement a pipeline of MATs, or anyway, they only provide little auxiliary functions.

In this chapter, we present the results of our efforts in providing the research community with a useful tool for the rapid prototyping of MAT-based forwarding abstractions. In a nutshell, our work consisted in accelerating OfSoftSwitch (OFSS) [3], and in demonstrating that such implementation can be effectively used to implement new MAT-based forwarding abstractions.

OFSS provides a linear and easy way to modify the implementation of the OpenFlow specification, which allows a programmer to quickly introduce new functions. However, it was not originally designed for performance, therefore an out-of-the-box OFSS can achieve only a very limited forwarding throughput of few thousands of packets per second (pps). We refactored a large portion of the OFSS' packet handling code, applying well-known techniques such as batching, zero-copy, static memory allocation, etc. Furthermore, we introduced PFQ [19] as packet I/O library to accelerate packets receive and transmission operations. Contrary to other packets I/O libraries, PFQ provides some auxiliary functions that help a programmer to quickly scale functions without dealing with problems such as packet dispatching to CPU's cores. Our accelerated OFSS (aOFSS) implementation keeps the original OFSS simple code structure but achieves a throughput of around 4.5 million pps (Mpps) on a single core. Furthermore, it can scale on multiple cores, achieving nearly 10Gbps line rate with minimum size packets when using four cores.

To demonstrate how our implementation choices can effectively help the implementation of new MAT-based abstractions, we also implement OpenState on top of aOFSS. OpenState is a recently proposed *stateful* MAT-based abstraction that realizes Finite State Machines (FSMs) in the data plane. Using aOFSS's OpenState implementation, we demonstrate a stateful firewall use case and compare its performance with the one achieved by Linux's Netfilter/iptables. While being a fast prototyping tool, our aOFSS implementation of an OpenState-based firewall provides more than 6x throughput improvement when compared to Linux's iptables.

The rest of this chapter is organized as follows: Sec. 5.1.3 presents OFSS architecture and issues. Sec. 5.1.4 describes the solutions and optimization performed in our implementation while Sec. 5.1.7 evaluates the achieved forwarding performance.

Open source contribution: aOFSS has been merged as BEBA-EU branch in the official OFSS repository and is already available at `https://github.com/CPqD/ofsoftswitch13/tree/BEBA-EU`

5.1.2 Related Work

To experiment with MAT abstractions, a possible way is to describe a MATs pipeline using the P4 [24] language. The P4 description is then compiled to a software implementation. To the best of our knowledge, PISCES [96] is the only high-performance implementation of a software switch whose MATs pipeline can be configured with P4. However, PISCES does not fully support P4 constructs. Most notably it does not implement stateful constructs. In fact, supporting stateful actions in a MATs pipeline abstractions is still an open research problem [60].

SoftFlow [60] addresses such problems extending OpenVSwitch to integrate more flexible processing blocks called SoftFlow actions, which can implement complex stateful functions. Combining SoftFlow actions with an OpenFlow-like pipeline of MATs enables a developer to perform arbitrary network functions. However, each SoftFlow action is, in fact, a black box, which consumes and produces packets, i.e., as if it were a VM attached to an OpenFlow switch. Click [79] is another tool for fast prototyping of network functions. Click adopts a model in which arbitrary functional blocks, called elements, can be composed into graphs. NetBricks [86] defines as abstraction a set of fine-granular primitives that suitably combined can describe some software network functions. However, the NetBricks' model is not specifically tailored for the implementation of MATs.

Differently, from the cited works, we provide an implementation of a MATs pipeline that can be easily modified to implement arbitrary network functions and experiment with variations of such abstraction. SoftFlow, Click, and NetBricks do not implement a MATs pipeline. For instance, a similar approach in SoftFlow would require modifications to the OVS code, whose complexity and issues we already discussed in the introduction of this chapter.

To accelerate packets I/O, we use PFQ [19] framework. Many other frameworks are currently available and an extensive comparison of them can be found in [26, 51, 77]. Relevant ones are PF_RING [50], PF_RING ZC (Zero Copy) [39], Netmap [93], DPDK [42]. PF_RING ZC, Netmap and DPDK bypass the Operating System by memory mapping the ring descriptors of NICs at userspace, allowing even a single CPU to receive 64 bytes long packets up to full 10 Gbps line speed. Also, DPDK adds a set of libraries for fast packet processing on multi-core architectures for Linux. Netmap and DPDK have been successfully used in accelerating software switches as in the case of the VALE [94] switch and mSwitch [56] (netmap) and CuckooSwitch [116] and DPDK vSwitch [58] (DPDK). Netmap was also used to accelerate packet forwarding in Click [92] and ClickOS [70]. PFQ, instead, relies on vanilla device drivers and leverages different levels of parallelism to accelerate packet I/O. Besides, PFQ is equipped with a native functional language to program in-kernel early stage packet processing.

5.1.3 OfSoftSwitch

OfSoftswitch (OFSS) is a userspace software switch implementation of the OpenFlow 1.3 specification. It is a popular tool in the academic community, as it provides a clean and flexible implementation of a MATs pipeline, which makes it suitable for functional experimentation. However, OFSS was not originally designed for performance, limiting its applicability as a rapid-prototyping platform to very basic functional evaluations.

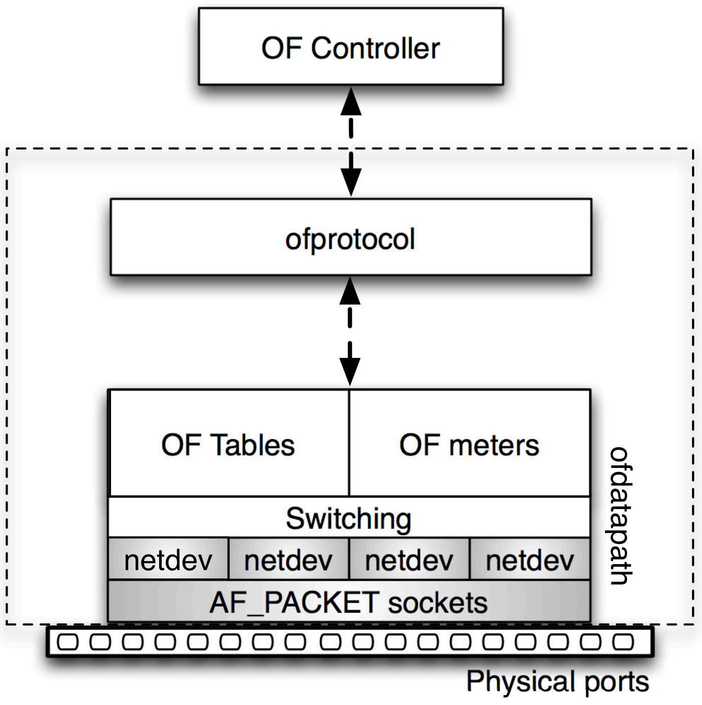


Figure 5.1: *OFSS architecture*

E.g., our tests show that OFSS can only forward few thousands of pps (cf. Sec. 5.1.7).
Architecture and Issues The architecture of OFSS is shown in Fig. 5.1 and reflects the OpenFlow architecture. Data plane and control plane are handled by two distinct processes: `ofprotocol` handles general configurations and the communication with an external controller; `ofdata-path` implements the switch's data plane. The `ofdata-path` module is designed as a single process application and relies on the `netdev` library to access network devices.

The `netdev` library implements an abstraction layer for network devices. As such, it contains all the functions for opening/closing devices, receiving/transmitting packets, managing queueing disciplines, reading devices' stats, etc. At the link layer, `netdev` relies on standard Linux `AF_PACKET` sockets for receiving and transmitting packets.

Two main issues impact performance in this architecture:

- the use of `AF_PACKET` sockets is well known to be inefficient regarding I/O speed.
- the switch runs in a single process/thread; hence, it cannot scale to multi-core processing.

5.1.4 Software acceleration

As a first step to improve the data plane performance of OFSS, we decided to replace the underlying `AF_PACKET` sockets for packet I/O operations. This required the modification of the `netdev` abstraction in order to enable the use of software accelerated frameworks such as PF_RING ZC [50], netmap [93], DPDK [42] or PFQ [23]. In fact, these frameworks introduce mechanisms for kernel bypassing, which help to avoid the performance bottlenecks of the operating system's network stack.

Each of the above acceleration engines would require a specific implementation of the new `netdev` library. Therefore, we decided to unify the different approaches by supporting the standard `pcap` library [89], a cross-platform interface for packet I/O supported by all of the aforementioned accelerated engines. As a result, the new OFSS's data-path is platform independent and can be transparently bound to any underlying accelerated engine supporting the `pcap` interface.

5.1.5 Multi-core processing

The I/O acceleration could be provided by any of the several frameworks we previously mentioned, because of the adoption of the `pcap` interface. However, supporting multi-core processing would still require a major refactoring of the OFSS's data-path. A refactoring we want to avoid for several reasons, including the ability to support, with little effort, already existing prototypes built on top of OFSS.

We found a solution to our problem using a particular set of features of the PFQ framework, which therefore became the framework of choice for our implementation.

The architecture of PFQ is shown in Fig. 2.1 of the chapter 2.

In this context, PFQ is used as an engine to retrieve packets from *one or more* traffic *sources*, make some *computations* by means of functional engines (the λ_i blocks in the picture) and finally deliver the processed packets to one or more *endpoints*. Traffic sources are either represented by Network Interface Cards (NICs) or – in case of multi-queue cards – by single hardware queues of network devices.

The selection of PFQ is motivated by its ability to enable fine-grained parallel computation in a simple and programmable way. While it is in principle possible for other frameworks to support similar functions, PFQ already integrates an in-kernel processing stage that is fully programmable through a high-level functional language. Such a processing engine works as a "pre-processing stage" and allows the execution of *dynamic* and *hot-swappable* (i.e., atomically upgradable at run-time) computations. As a result, packets can be filtered, logged, forwarded, load-balanced and dispatched on a per-packet basis to *groups* of application sockets, to generic endpoints or even to the kernel.

PFQ integrates several primitive functions that can be roughly classified as: protocol filters, conditional functions, logging functions, forwarding (to the kernel or NIC) and fanout functions (mainly, steering). Steering functions are particularly relevant to enable parallelism as they allow an application to deliver packets to a group of sockets by using a hash-based *stateless* load balancing algorithm. Both the algorithm and the hash keys are defined through the computation instantiated in the functional engines. For example, the function `steer_flow` spreads traffic according to a symmetric hash that preserves the coherency of bi-directional flows, while the function `steer_ip` steers traffic according to a hash function that uses source and destination IP addresses. More generally, steering can be performed according to arbitrary criteria with the overall target of distributing the processing and avoiding state sharing across cores.

Such feature turned out to be straightforwardly applicable to enable the scaling of *stateless* OFSS application, such as the standard OpenFlow implementation, but also very useful to support *stateful* applications. More specifically, Fig. 5.2 shows

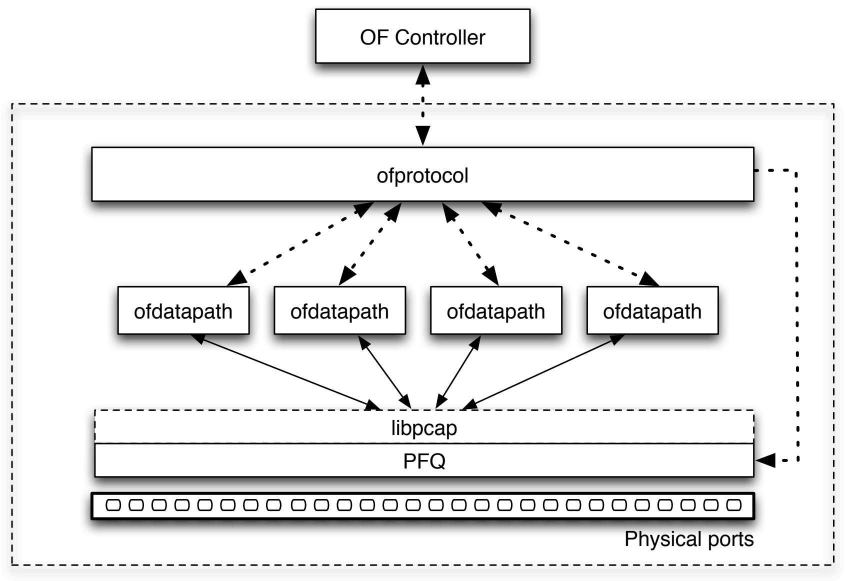


Figure 5.2: *Multi-process OFSS*

the multi-core processing scheme we implemented for OFSS. Notice that the multiple `ofdata-path` instances (workers) act as independent switches that operate on disjoint portions of the overall traffic. Using the steering function of PFQ, the traffic is distributed across different workers. Applications can scale to multiple cores by configuring the PFQ functional engines according to the application's specific needs. Such operation can be easily performed in the `ofprotocol` module.

For example, stateful abstractions, such as OpenState [16] and FAST [80], separate state access during packet processing on a per-flow basis. A *lookup* operation at the beginning of the MAT identifies the packet's flow and, consequently, the state that will be accessed and modified during the packet processing. Such lookup operation can be used to directly derive the PFQ steering criteria (keys), which in turn guarantees that PFQ will dispatch a packet to the process that keeps the state required for the processing of such packet.

5.1.6 Code Optimizations

Although I/O acceleration was the main direction to improve the OFSS data plane performance, a thorough code analysis of the switch implementation revealed a significant number of additional performance bottlenecks. All such bottlenecks have been extensively investigated, and quite a few sections of the original code have been redesigned and optimized according to more efficient network programming principles. Later, a list of the major performance modifications to the original software architecture is reported.

Dynamic Memory Allocation. The data-path of OFSS makes extensive use of dynamic memory allocations and related memory releases. This dramatically impacts the packet forwarding performance as the cost of each pair of calls is around 200-500 CPU cycles. We implemented a *zero-malloc* optimization that allows OFSS to run without performing dynamic memory allocations. Whenever required, the semantics of the data structures have been changed to cope with memory buffers without ownership, which, in turn, are passed along the data-path as managed memory. Furthermore, the packet handler has been re-designed to fit into a single chunk of memory, replacing the original scattered model. This permits to save two extra additional memory allocations and de-allocations.

Hash Maps Refactory. Hash maps are pervasively used throughout the OFSS data-path. Wherever possible, hash maps have been replaced with more efficient `struct` data types to save very frequent memory indirections when accessing specific protocol fields. Also, the remaining hash maps have been equipped with a set of managed small memory nodes, which are allocated at construction time.

Since hash tables are aware of whether the memory nodes are managed or not (using an annotation on every single node), they can concurrently use both the small set of pre-allocated nodes and the additional nodes allocated on-demand. In the case of the hash tables associated with the packet handler, this optimization allows saving (on average) up to 3 or 4 table rehashes, as they systematically host around a dozen of entries per each received packet.

Zero Copy. Both the semantics of `pcap` and PFQ allow one to take advantage of the memory persistence of a packet, during the call of a `pcap` handler. This semantic has been leveraged to retain from saving a copy of the payload of each packet, when not strictly required[1]. Strictly speaking, the *zero-copy* optimization consists in removing a pair of `malloc/free` together with a `memcpy`.

Batch processing. The original version of OFSS processes one packet at a time. Instead, we enabled batch processing of packets. Therefore, the forwarding function of OFSS has been changed to consume, per each port, a batch of packets up to a configurable number, before switching to another port. The beneficial effects of such an optimization are mainly due to the increased cache locality that occurs while processing packets. Also, in modern CPUs, this mode of operation allows one to take advantage of packet pre-fetching. That is, the CPU is explicitly instructed to pre-fetch data while doing some other processing. As a result, the CPU can retrieve one or more consecutive packets while the current one is being processed.

5.1.7 Performance Evaluation

We carried out an extensive experimental campaign, under different scenarios, to understand the absolute performance of our implemented prototype and its scalability. The experimental test bed consists of two identical machines with 8-core Intel Xeon E5–1660V3 CPUs (3.0GHz), equipped with a pair of identical Intel 82599 10G NICs. One of the machines runs the software switch, the other is a load generator. Both systems run a Linux Debian stable distribution (kernel v. 3.16). A third server runs the controller and is connected to the switch's server using 1G control network interface.

OpenFlow performance. The first set of experiments are pure *speed tests* to benchmark the performance of the accelerated version of OFSS (aOFSS) when running a standard OpenFlow pipeline. Figure 5.3 shows the achieved throughput when varying the number of cores and the packet sizes. Both the original OFSS performance and line rate limits are also reported for comparison. The implemented acceleration techniques

[1] E.g., when the packet is consumed in the contest of the forwarding thread of execution.

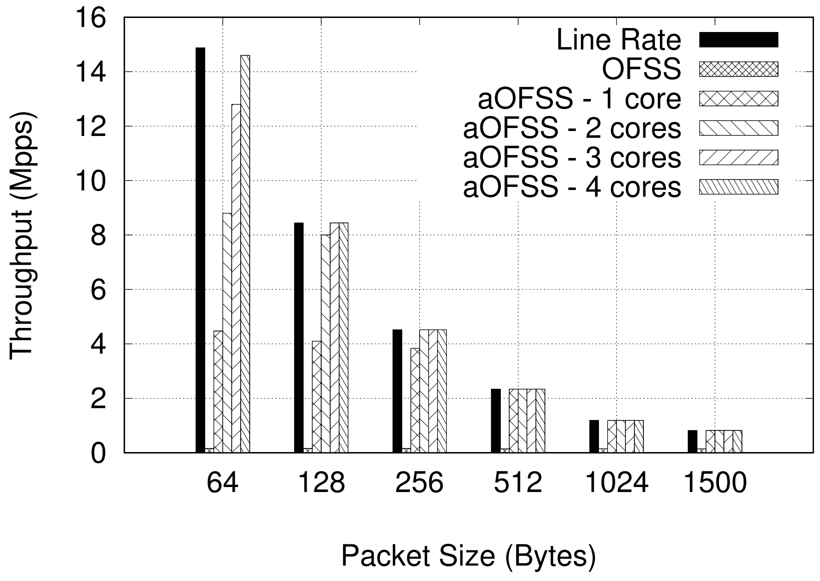


Figure 5.3: *OpenFlow pipeline throughput*

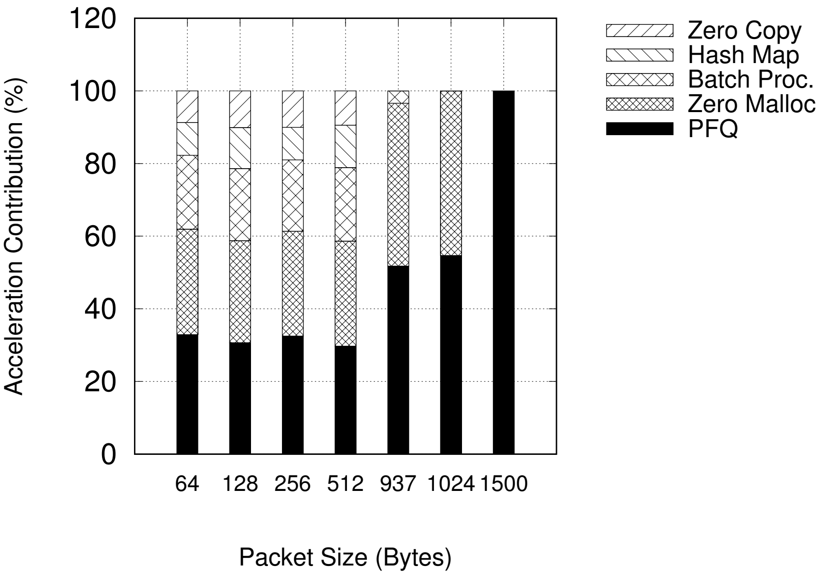


Figure 5.4: *Acceleration contributions to OFSS*

provide a dramatic performance improvement, with the throughput nearly hitting line rate in all the cases and up to 96x speedup factor concerning the original OFSS.

The contribution of each optimization technique to the total throughput is reported in Figure 5.4. The results are obtained by selectively switching off one contribution at a time on the accelerated version of OFSS running on a single core and measuring the observed performance drop. Data are finally normalized to give a fair visualization of each term as a stacked histogram. It is worth noticing that, in this case, multi-core acceleration is not accounted since the experiment was run on one core. The results show that the generic I/O acceleration provided by PFQ and the zero-malloc optimization have a mostly equally beneficial impact on the performance boost. For shorter packet sizes, the impact of the other optimization is all but negligible as they contribute for up to 40% of the overall performance improvement.

OpenState performance. The second set of experiments measures the aOFSS performance when doing *stateful* operations. In particular, we adopt an OpenState prototype that was originally built on top of OFSS. Since OpenState can build a pipeline out of both *stateless* and *stateful* MATs we evaluate both cases. Furthermore, we also evaluate aOFSS performance when the depth of the pipeline changes.

Fig. 5.5 shows the throughput achieved by aOFSS when using *stateless* OpenState stages. The system still hits line rate for packets of more realistic sizes of at least 128B. In the most critical case of shortest packet size, performance decreases with the number of stages, but still reaching well above 10 Mpps with four running cores. Notice that the performance for one stage is comparable to the ones of OpenFlow. In fact, a stateless stage is functionally equivalent to an OpenFlow MAT.

Fig. 5.6 shows the performance for a pipeline of *stateful* OpenState stages. As expected, the performance decreases, with line rate achieved for packets of at least 256B size. The degradation is more significant as the number of stages increases. However, we remark that in our test we measured a somewhat worst-case behavior. That is, *every packet performs a change to the switch's state*. For reference, such action, in fact, corresponds to changing the forwarding entry handling the packets, for each received packet.

Finally, using the approach presented in [87], we implemented a stateful firewall function cascading two stateful OpenState stages. The function is equivalent to the one provided by a Linux `iptables/netfilter` firewall. In such a setup, and when processing 65K network flow, our implementation could achieve 8.2 Mpps forwarding throughput with minimum sized packets. For comparison, we measured the performance of a Linux system's network stack, when providing the same function. In particular, we configured iptables with the following rules:

```
iptables -A FORWARD -i e2 -o e1 -j ACCEPT
iptables -A FORWARD -i e1 -o e2 -m state
         -state ESTABLISHED -j ACCEPT
```

Linux's iptables could achieve a 1.3 Mpps throughput when handling the same flows. I.e., the aOFSS implementation of a firewall is more than 6x faster than a vanilla Linux implementation[2].

5.2 PFQ as measuring instrument

5.2.1 Introduction

It has been reported that TCP traffic represents 80-90% of the packets and bytes flowing today through the Internet [28]. It follows that most of the traffic sources adapt their sending rate according to the perceived available bandwidth. Indeed, TCP is the

[2]For a fair comparison; we configured the system in similar ways when doing the two tests, i.e., four cores to perform the switching and forwarding operation in both cases. Also, notice that iptables implements the stateful firewall relying on the Linux's `CONNTRACK` module.

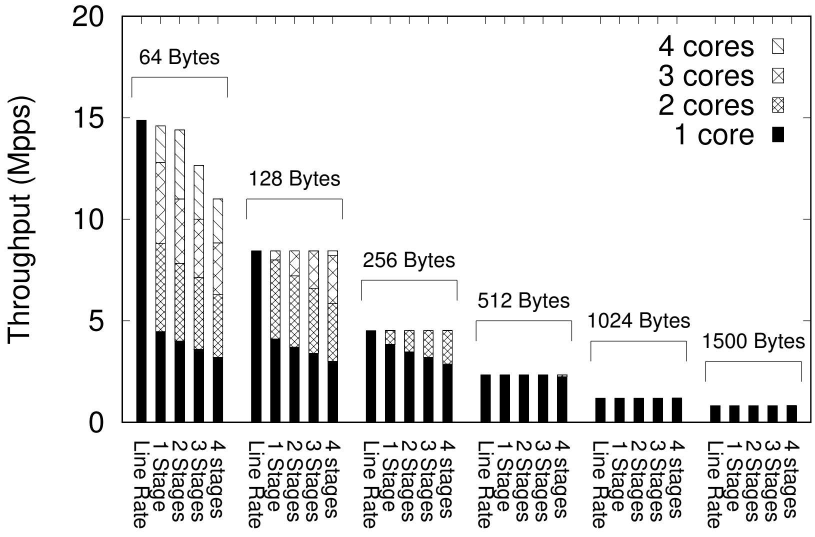


Figure 5.5: *Stateless OpenState stages throughput*

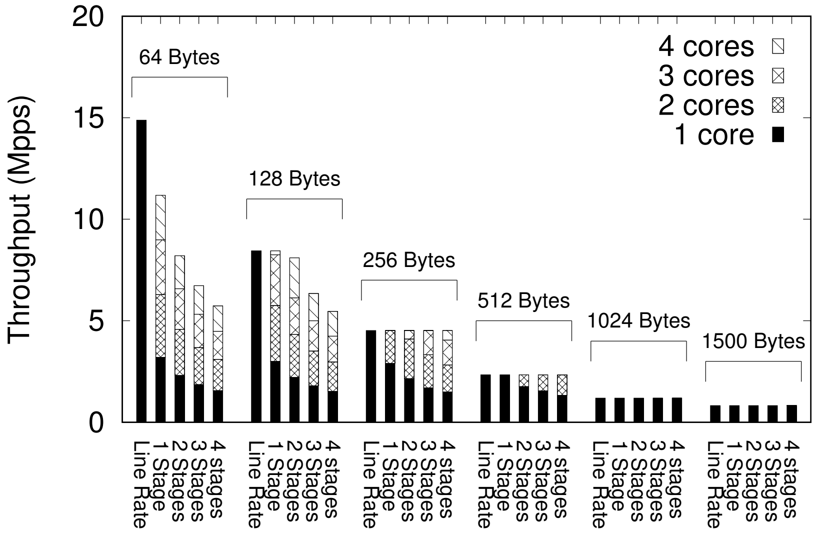


Figure 5.6: *Stateful OpenState stages throughput*

instantiation of an important design choice that contributed to the success of the Internet: to leave congestion control to the end-systems, thus permitting a relatively simpler implementation of the interconnection devices. TCP rate control algorithms, such as Additive-Increase-Multiplicative-Decrease (AIMD), help maintain a fair allocation of network resources on a *per-flow* basis. In the simplest case of multiple TCP streams, all experiencing the same RTT and sharing the same FIFO queue, each flow tends to occupy the same portion of the link bandwidth [91].

However, relying only on end-systems to guarantee fairness is not enough due to ill-behaving users and issues intrinsic to TCP-like algorithms. Examples of cases of unfairness are: (i) applications that open a large number of parallel TCP connections, e.g. peer-to-peer, or that tweak TCP to get better performances; (ii) non-TCP-like protocols, i.e. protocols that do not respond to congestion signals such as drops, and (iii) the dependence of standard TCP to the round-trip times (RTT) [91].

For these reasons, most Internet service providers (ISPs) tend throttle customer traffic at the network edge, limiting the maximum bandwidth of each user to a feasible, but *static* network allocation. This approach allows ISPs to leave their core and interconnections with other ISPs uncongested at all times. The downside is that the excess bandwidth remains unused, even in common situations of low usage, such as at night.

Researchers have proposed solutions to enforce a more *dynamic* bandwidth allocation in the network interconnection devices. In these approaches, instead of capping the maximum sending rate at all times, network devices can redistribute the unused capacity (if any) to those users asking for more. The trick here is to design a bandwidth enforcement scheme that (i) guarantees that all users can obtain at least the level of service they paid for, i.e., minimum rate guarantees, and (ii) when unused capacity is available, that is shared by all users, with no one prevailing on others. Ideally, such mechanism should be introduced in the network without compromising today's line rate requirements, i.e., 10-100 Gbit/s per port.

Fair Queuing (FQ) scheduling [38, 81] is the textbook approach to enforce almost perfect fairness among different traffic sources, independently of the behavior of the end-hosts. A switch implementing FQ works by assigning users to different queues, where a "user" is an arbitrary aggregate of packets, e.g., with the same IP source address or the same TCP/UDP 5-tuple. FQ provides high precision of bandwidth partitioning but, unfortunately, such precision comes at a considerable expense: (i) the time to process a packet depends on the number N of active users, precisely $O(\log(N))$; and (ii) N per-user queues are required.

The first limitation is important with today's throughput requirements which drastically reduce the maximum processing time allowed for a packet, e.g., a switching chip with aggregate throughput of 1 Tb/s has a time budget of 1 nanosecond to process a minimum size packet. The second limitation affects switching hardware implementations. Here the number of queues impacts both the memory requirements and the combinatorial logic necessary to implement the scheduler circuitry. Indeed, for a scheduler to be work-conserving, i.e., to serve a packet if at least one can be served, all N queues must be examined at the same time. Thus, the number of wires to implement such a structure depends on N. As a consequence, it is hard to scale FQ implementations to hundreds, thousands or more users. For this reason, the number of queues available in commercial hardware switches is usually bounded to less than ten [6]. This consideration is also at the base of legacy quality of service (QoS) approaches such as DiffServ, where traffic is aggregated into few classes.

In this study, our focus is to devise a design for a bandwidth enforcing scheme in which both time and implementation complexity do not depend on the number of active users N.

This work is inspired by recent advances in Software-Defined Networking (SDN) and data plane programmability. Emerging abstractions such as P4 [24], OpenState [16], OPP [17], FAST [80], and Domino [99] allow network operators to perform flexible stateful packet processing inside the network. The statefulness of the aforementioned approaches lays in the ability to program forwarding rules that read and modify data plane's forwarding state. Based on this capability, some studies have been published, showing how to implement existing and new forwarding functions using programmable data planes [32, 64, 97, 101].

We follow this path and design a scheme to enforce fair bandwidth sharing that is amenable with programmable data plane abstractions. To this purpose, we do not modify the scheduler, and we use, instead, a widely-deployed strict priority scheduler with only a few queues. Fairness is enforced by dynamically assigning priorities to users according to their sending rate history. We call our design FDPA (Fair Dynamic Priority Assignment). In FDPA, packets belonging to a user whose arrival bitrate is equal or less than its fair share are given priority over those users generating traffic at higher rates. FDPA does not provide precise bit-level or packet-level fairness, but it approximates a fair repartitioning over longer timescales, in the order of few RTTs.

The scalability of FDPA does not depend on the number of queues, but instead on the state available for the rate estimator. Precisely, while the circuitry to implement a rate estimator can be shared among many flows , the switch is required to maintain per-user state, i.e., the measured rate. Hence, the only limit of FDPA is the memory available in a switching chip.

We address the applicability of the FDPA approach by performing experiments on a 10 Gbit/s testbed using a software prototype implementation. Results show that FDPA produces fairness comparable to other schemes based on scheduling. However, we find that FDPA introduces a trade-off between fairness and throughput, in which one or the other are penalized.

To summarize, the contributions of this chapter are:

- Design of FDPA, a scheme to enforce approximate fair bandwidth sharing among many users. Switch requirements to support FDPA are a (i) strict priority scheduler and (ii) the ability to manage data plane's state to measure the arrival bitrate of each user.

- Evaluation of FDPA and other Linux's traffic management schemes using a 10 Gbit/s testbed with real TCP traffic.

We begin by reviewing the related work in subsection 5.2.2, we then introduce the FDPA design in subsection 5.2.3 and discuss its implementation options with programmable data planes. In subsection 5.2.4 we present the experimental results from the 10 Gbit/s testbed, before concluding with a discussion on open questions and future work in subsection 5.2.5.

5.2.2 Related work

To reduce implementation and time complexity of FQ, some algorithms have been proposed in the literature. Deficit Round Robin (DRR) [98] is probably the most known and widely-deployed one. DRR was proposed to address the time complexity of FQ. Indeed, DRR achieves $O(1)$ execution time per packet. However, DRR still requires per-user queues, greatly limiting the maximum number of distinct users that can be served by the scheduler.

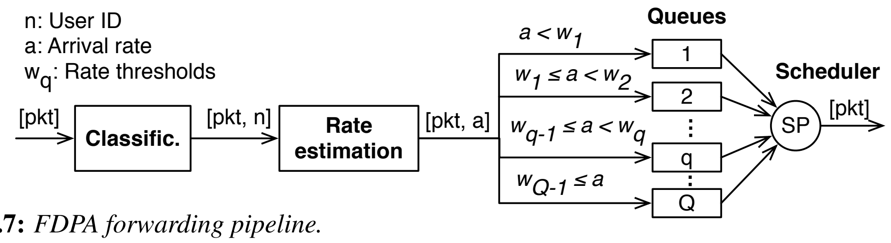


Figure 5.7: *FDPA forwarding pipeline.*

To overcome DRR's limitations, further approximations have been proposed. Stochastic Fair Queuing (SFQ) [71] is a probabilistic variant of FQ. Here traffic streams are hashed onto a smaller number of queues, and the hash function is periodically perturbed to minimize the time where two users collide onto the same queue. Here the quality of the approximations depends on the number of queues, and the perturbation interval. Finally, Approximate Fair Dropping (AFD) [84] employs a form of active queue management (AQM) by dropping packets before being stored in a simple FIFO queue. Dropping decisions are based on the recent history of packet arrivals, with a higher probability of drop for users sending at higher rates. AFD has been used in several switch and router platforms at Cisco Systems [85].

Our approach shares the same design principles of AFD: (i) avoid using per-user queues in favor of the per-user soft state, and (ii) achieve bandwidth partitioning by opportunistically dropping or delaying packets rather than enforcing rate by using scheduling. However, while the AFD design allows for an efficient implementation in a fixed-function ASIC, its realization with programmable data plane primitives might not be straightforward. Specifically, AFD requires the implementation of a shadow buffer in which packets are removed at random. We are not aware of any data plane abstraction providing native support for such data structure. Its behavior could be approximated using other primitives, however, this would require a dedicated study. Instead, we prefer to explore the feasibility of FDPA which, as will be discussed in subsection 5.2.3, requires much simpler primitives exposed already by current data plane abstractions.

Finally, a more recent approach named PIFO has been proposed to address the need of a programmable scheduler [100]. However, similarly to fixed-function schedulers, in PIFO the number of distinct flows that can be served with a fair queuing discipline is bounded by the number of queues. In their proposed design, such bound is 2048 in total or 32 per port in a 64 port switch. While one could imagine dedicating all 2048 queues to a port, the authors do not provide any evaluation of their scheduler with realistic traffic traces.

5.2.3 FDPA Design

In this subsection we describe the design of a packet forwarding pipeline implementing FDPA. To simplify the exposition and without loss of generality, we assume a switch with rate controlled only on one egress port.

Figure 5.7 depicts the design of the pipeline. Packets are first classified per user and then processed by a rate estimator which measures the arrival bitrate of the specific user. Packets are then stored in one of the Q priority queues such that the higher is the arrival rate, the lower will be the priority. A strict priority scheduler (SP) serves queues in priority order: packets of priority q are dequeued only if all other queues with higher

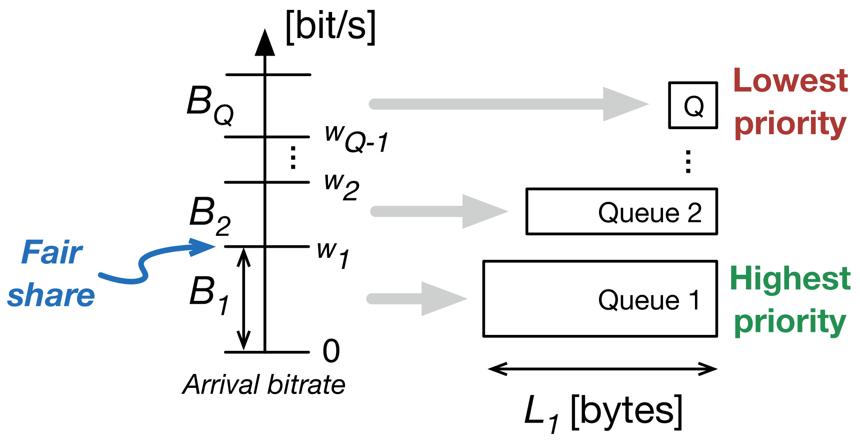


Figure 5.8: *Rate bands and queue size in FDPA*

priority are empty, where $q = 1$ is the highest priority.

The measured arrival rate for a given user at a given point in time determines an active *band* for that user. Packets arrived in band B_q will be assigned with priority q (Figure 5.8). The first band B_1 represents the minimum guaranteed portion of the link capacity allocated to each user, for this reason, B_1 should be dimensioned such that $N \times B_1 \leq LinkCapacity$. Moreover, to further penalize ill-behaving users, each queue has a different size L_q, with smaller values for low priority queues.

Rationale

To discuss the rationale behind this design, we begin with the case of a scheduler with only two queues ($Q = 2$), high priority and low priority; we then explain the need for more queues.

Two priorities. When congestion occurs, users sending below their fair share are prioritized against others sending at higher rates. Packets with low priority are delayed and in the worst case of a full buffer, dropped upon arrival. Such an event signals the TCP source to reduce the transmit rate. With FDPA, this reduction is expected to continue until the transmit rate hits the first band, in which case the user is prioritized again. Assuming that all sources are TCP-like and produce long-lived flows, under severe congestion, we expect traffic sources to shape their transmit rate around their fair share, i.e., the upper threshold of B_1.

Unfortunately, swapping queues can frequently cause packet reordering at the receiver, confusing TCP congestion control and affecting throughput. The problematic part is when users are prioritized again, i.e., their assigned queue is changed to the one with high priority. Here the same burst of consecutive packets might be stored first in the low priority queue and then in the high priority one, with the effect of having subsequent packets being transmitted before those arrived earlier. We are interested in measuring this effect when using FDPA.

In the case of non-elastic sources, e.g., constant bitrate, B_1 represents the maximum rate that a source can send with guarantees of bounded latency and minimum drop probability. Indeed when a user hits the first band, packets are always served by the same, maximum priority queue, hence preventing disruption from other TCP sources aiming to transmit at higher rates.

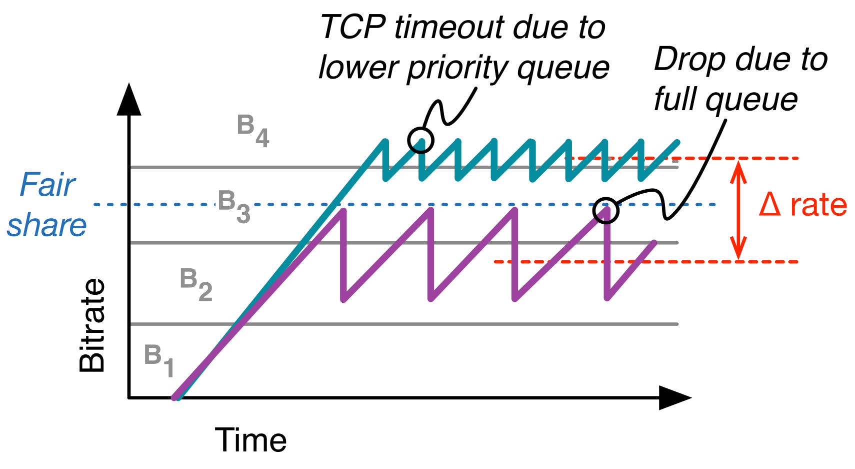


Figure 5.9: *Example of 2 TCP sources competing for the excess bandwidth when using more than 2 priorities.*

However, if some sources are using less than their fair share or because not all the link capacity has been reserved, i.e., $N \times B_1 < LinkCapacity$, using only two priorities does not enforce equal distribution of the excess bandwidth. Indeed, if we assume that capacity has been allocated for many users, but only a few of them are active and sending TCP traffic, we can expect that those users will be competing in the same low priority FIFO queue, without any guarantee of fairness.

More priorities. To enforce equal distribution of the excess bandwidth, we need to introduce more priorities, such that the more a source increases its sending rate, the lower will be the priority compared to other users. When all sources are TCP-like, following the same rationale of the previous case, we expect the transmit rate of each user to converge to a fair share that considers the excess bandwidth. Such fair share will lay in a rate band other than B_1

Figure 5.9 illustrates the expected behavior of 2 TCP-like sources competing for the excess bandwidth. In this example, one source (1) is ill-behaving as it uses a more aggressive rate control algorithm (similar to the case of a user opening multiple TCP streams); the other source (2) is well behaving, as for each congestion signal it halves its transmit rate. At steady state, both sources tend to share the same queue with priority 3, however, the different rate-control behavior that they implement causes them to oscillate around different average values. Indeed, (1) always tends to increase its rate until it falls in the 4th band, which causes its packets to timeout as the scheduler will spend as much time as needed to serve packets of higher priority; (2) instead has higher drop probability when it falls in band B_3, as here the queue is monopolized by packets of (1). However, by always assuring a higher priority for lower rates, the increase of (2) is always guaranteed at least until the lower threshold of band B_3. Intuitively, we expect that the difference between the average transmit rate ($\Delta rate$) will be smaller with narrower bands, hence producing a more fair allocation.

Unfortunately, as in the case with only two priorities, we expect that multiple narrower bands will increase the risk of packet reordering, affecting the overall throughput. We are interested in measuring such a trade-off between fairness and throughput.

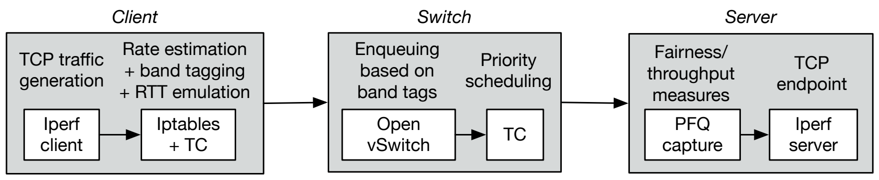


Figure 5.10: *Software-based processing pipeline used in experiments.*

Implementation with programmable data planes

Classifying packets per user is easy and can be done using a match-action table as defined by OpenFlow [72] or P4 [24]. Using such tables, one can match on specific header fields and write the corresponding user ID n on the packet's metadata.

Estimating the bitrate of a flow might be tricky at line rate. In the simplest case, the switch needs to maintain for each user a byte counter and a timestamp of the last time the rate estimation was updated. Updates of the rate values are triggered by packets arrival if the timestamp of the packet exceeds a predefined interval, i.e., the minimum interval over which the average bitrate is evaluated. The rate is then computed dividing the number of bytes by the interval between the packet's timestamp and the stored timestamp. While the division is an operation that might be hard to perform in a line rate switch, in [97] it is shown how this operation can be approximated with reasonable precision using look-up tables available in programmable data planes. A second match-action table can then be used to direct packets to the different queues according to the estimated rate band, written in the packet's metadata.

Along with programmable data planes, FDPA can be implemented in switches supporting OpenFlow v1.3+. Indeed, OpenFlow defines “meters” that can be configured with different bands as defined by FDPA, such that packets hitting a given rate can be marked using the DSCP field.

Finally, priority schedulers are a standard component available in today's switching hardware.

5.2.4 Experimental results

We now evaluate the feasibility and performance of FDPA using a software-based prototype implementation. We are interested in measuring the effects of different band assignment on both fairness and throughput. We also compare FDPA with other approaches such as DRR.

Testbed

We used three desktop machines with 8-core Intel Xeon E51660V3 CPUs (3.0GHz), equipped with multiple Intel 82599 10GbE NICs. One machine acts as a switch with four 10 Gbps ports, another one is used to generate traffic from two ports, while the last is used to both produce and receive traffic from different ports. Each machine runs a Debian 9.0 Stretch based on a Linux Kernel v4.9.16.

Figure 5.10 shows the processing pipeline used to emulate FDPA. We use `iperf` to generate TCP traffic, Linux's `iptables` to estimate the rate and tag packets accordingly. In our design, rate estimation should happen in the switch, however, to simplify the prototype implementation we decided to move it to the client machines. We use Linux's `tc` (Traffic Control) to emulate different RTTs at the clients and to perform priority scheduling at the switch. Open vSwitch is used to steer packets to the different queues based on the band tags. Finally, we use PFQ [21], a framework for accelerated packet I/O, to measure the bitrate of each user. Both clients and server use TCP Cubic, with the default parameters found in the Linux Kernel v4.9.16. We only adjust the memory available to TCP buffers to allow for a large number of connections. We set the MTU of all interfaces to 1500 bytes.

We configure sources to experience an emulated RTT of around 5 ms with maximum 0.25 ms of variable jitter with 25% correlation. TCP increases its sending rate at RTT timescales, hence for FDPA to promptly respond to rate variations, the estimation interval should be in the order of few RTTs. For this reason, we set the estimation interval to 30 ms.

Metrics

We measure the quality of an experiment using two metrics: (i) the aggregate throughput (TPut) normalized over the link capacity, i.e. bounded between 0 and 1, and (ii) the Jain's Fairness Index (JFI) [62]. The JFI is a popular fairness measure defined as:

$$JFI = \frac{(\sum_n x_n)^2}{N \cdot \sum_n x_n^2}$$

where x_n is the normalized rate of a user n and N is the total number of users. The normalized rate is defined as $x_n = MeasuredRate_n / FairRate_n$. In our experiments each user is assigned with the same fair share, i.e. $FairRate_n = LinkCapacity/N \ \forall n = 1...N$. The JFI is bounded between 0 and 1, where 1 is a fair distribution and 0 is a discriminating one. In testing FDPA we aim at maximizing both TPut and JFI.

Results

Figure 5.11 shows the results obtained from the experiments. We generate long-lived TCP traffic varying the number of users to 50, 100 and 200,[3] and varying the number of TCP connections per user based on four scenarios: (i) all users open only one TCP connection,

i.e., they all well-behave, (ii) 25% of the users misbehave by opening ten parallel TCP connections, while the remaining 75% only 1 (iii) 50% of them misbehave, and (iv) the number of connections per user is uniformly distributed between 1 and 10.

We also vary the number and size of rate bands. We use the following notation to describe an FDPA configuration: $F(FirstBand + NumBands * BandSize)$, where $FirstBand$ is the size of B_1, $NumBands$ is the number of bands following the first one, each one of size $BandSize$, except for the last one that has infinite size, i.e. up to the link capacity. $FirstBand$ and $BandSize$ are expressed as a proportion of the fair share, e.g. $F(1 + 4 * 0.5)$ describes a configuration where the first band is exactly the fair share, and the other 4 bands have size half of the latter. We perform experiments with $FirstBand \in \{0.75, 0.85, 1, 1.15, 1.25\}$, $NumBands \in \{3, 4\}$ and $BandSize \in \{0.25, 0.33, 0.50, 0.67, 0.75\}$. For sizing the queues we empirically found

[3]We put a limit to 200 as we noticed that our experimental setup suffers from performance degradation when emulating more users.

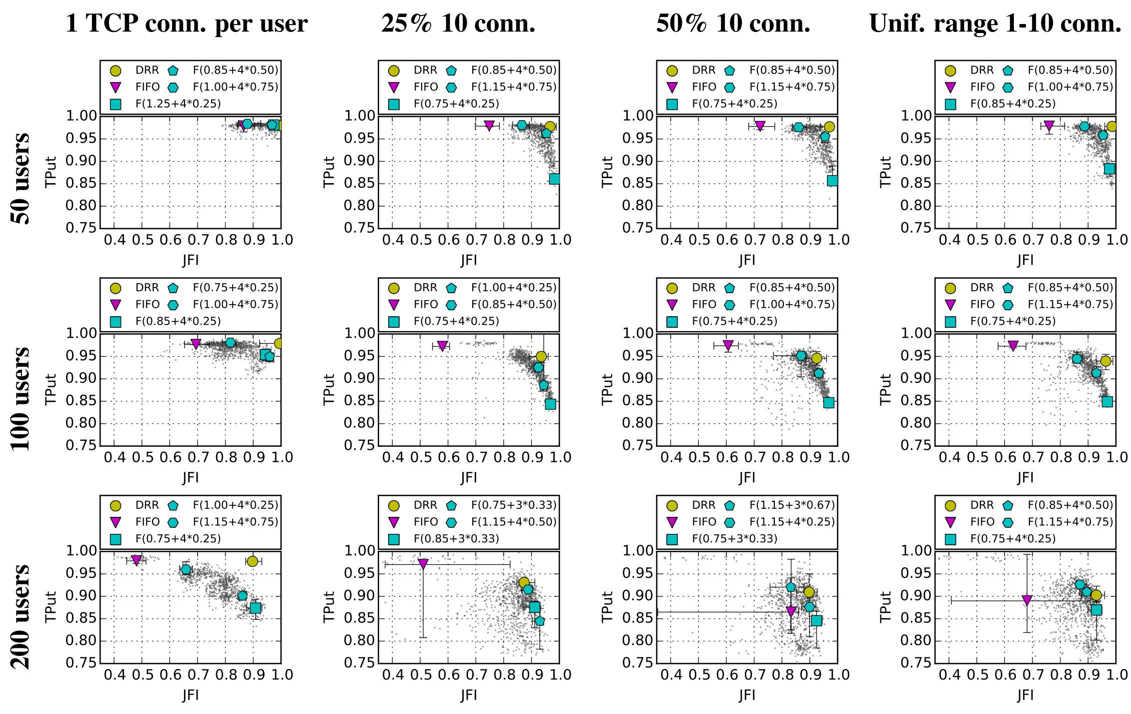


Figure 5.11: *Experimental results*

that the following rule provides optimal performances: $L_q = \min(20, \overline{BDP/q^q})$, where BDP is the bandwidth delay product $\overline{RTT} \times LinkCapacity$. With $\overline{RTT} = 5$ ms, the sizing for 5 queues is $L_1 = 4166$ MTU-size packets, $L_2 = 1041$, $L_3 = 154$, $L_4 = 20$, and $L_5 = 20$.

At the server, we collect samples of the average bitrate over a 1-second interval, each second at the same time for all sources, for 50 seconds. We start sampling 30 seconds after starting `iperf`, allowing all TCP sessions to converge to their average bitrate. For each second, we then compute both the JFI and TPut. In the plots, we show the median of the JFI and TPut samples for each experiment, along with an 80% confidence interval. For each traffic scenario, we plot only three configurations of FDPA, the one with the best TPut, the one with the best JFI, and the one that maximizes the product of both. We also provide a scatter plot of all JFI and TPut values obtained in all FDPA configurations. This explicitly shows the trade-off between TPut and JFI.

Finally, we compare results with the following cases:
FIFO. All users are served using 1 FIFO queue of size $L = BDP$, e.g. 4166 MTU-size packets with $\overline{RTT} = 5$ ms. This is our worst case when fairness is not enforced.
DRR. The switch implements DRR scheduling with per-user queues. We use the `tc-drr` implementation provided as part of the Linux's `tc` suite. We use DRR as the best case scenario; however this should be considered as an *ideal* case. Indeed, while it is still feasible to provide a large number of per-user queues in software, the same does not apply to hardware switches, where an a-priori instantiation of hardware resources (memory and logic circuitry) is required for each queue. The reader should remember that the majority of today's switching chips provide 10 or fewer output queues per port [6].

As expected, FDPA holds the promise of enforcing fairness w.r.t. a single FIFO queue in all scenarios, producing results comparable to the ideal case of a DRR scheduler with per-user queues. However, with FDPA fairness comes at the expense of throughput. We observe how configurations of FDPA that use narrower bands provide more fairness, between 0.95 and 0.99 in most cases. Unfortunately, these settings systematically incur in throughput degradation, down to 0.85 in some cases, while for the same scenario DRR achieves almost perfect fairness with throughput comparable to that of a FIFO queue, i.e., optimal around 0.98, or little less around 0.95. Vice versa, larger bands improve throughput, at the expense of fairness.

5.2.5 Discussion

How to improve throughput? Preliminary analysis shows that throughput degradation is mostly caused by packet reordering due to frequent changes in the queue assignment, which confuses the TCP congestion control. A solution to this problem could be that of using a *flowlet-based* approach [63], in which queue assignments are valid for the whole burst of packets, where bursts are separated by an idle time usually comparable to the RTT. This would decrease the probability of having back-to-back packets sent out from two different queues, and hence packet reordering. Detecting flowlets is a common function implemented by stateful data plane abstractions [31, 64, 97]. We leave exploring such a more advanced design for future work.
Rate estimation. An alternative to average estimators is the token bucket-based estimator. The advantage of using token buckets lays in their ability to immediately respond to rate spikes and bursts of packets, while an average estimator might leave enough time to an aggressive user to congest the highest priority queue. We know that the downside of frequent band variations is a higher risk of packet reordering, and preliminary results on our testbed using token buckets show that this is the case. However, we believe that using token buckets along with per-flowlet queue assignment could help in improving both fairness and throughput. We leave this for future work.
How to compute the fair share? We envision an external controller (or switch-internal control plane) that periodically adjusts band sizes by counting the number of active users. In the case of a service provider network, where the number of active users varies slowly, we do not expect that the frequency of the estimation process might be a limit for the scalability of the approach. Indeed, using many priorities helps in also absorbing minor variations of the fair share. How to efficiently implement user estimation is outside the scope of this study. However, we note that a controller could use the same counters instantiated at the switch for the rate estimation process.

5.3 Probabilistic counting framework

Traffic measurements and monitoring are routinely performed by network operators to assess the operational status of their infrastructure as well as to detect possible misbehavior and malicious activity. No matter of the ultimate goal, in the vast majority of the cases such investigations ultimately involve the primitive of *counting* packets or flows of packets having predefined characteristics. The application scopes can be manifold, from troubleshooting and sanity check, up to traffic profiling for commercial use, or anomaly detection for network security. In all cases, however, the process of retrieving statistics from real measurements is dramatically complicated by the ever-increasing link bandwidth that nowadays requires measurements to be taken on-the-fly on top of at least 10GB+ network segments.

In such network scenario, even an intuitively *easy* operation such as counting may become hard, as the time budget available to i) discriminate the data of interest first and to ii) select and increment memory registers easily drops below a few nanoseconds. Besides, high-speed links naturally induce likely large numbers of packets to be filtered first and counted later. This operation requires larger data structures which, in turn, must be placed in larger (but slower) DRAM memories. As a result, consolidated techniques, like those based on traditional hash tables, become practically unfeasible in many cases.

The whole frame gets further complicated whenever monitoring and measurements are carried out in a distributed fashion, i.e., by placing probes at multiple vantage points. In fact, this is now standard practice for network operators whose objective is to come out with higher-level aggregate statistics from single measurement collections. Cumulative statistics are typically performed by *mediators* in charge of analyzing and profiling traffic and possibly generate alerts and execute mitigation actions in case of detection of suspicious anomalies. At this stage, a big issue to be addressed is to prevent mediators from accounting duplicated flows monitored at different probes. This problem may be solved either by computational intense post-processing operations on all single traffic dump or (and this is the solution proposed in this section) by adopting data structures that automatically discard *by design* duplicated data when they get merged.

As a final remark, another big issue that network monitoring practice must meet is the compliance with current legislation regarding *privacy preservation*. In particular, due to current legislation trends (the paper [15] effectively explains the constraints imposed, for example, by EU legislation), much of the information retrieved from the captured traffic is considered privacy-sensitive, and its disclosure and storage is subject to strict rules. Such constraints not only applies to verbose packet traces, but a huge number of derived metadata (including per-flow counting reports) are considered to contain privacy-sensitive information and therefore their export and storage (if not wholly forbidden) is subject to stringent rules. As such, specific details on single flows should be disclosed only if strictly necessary (*necessity principle*) and with a level of granularity that must not be excessive concerning the purposes for which data are collected and further processed (*proportionality principle*). This last requirement calls for compact and reasonably "anonymous" data structures that keep aggregate statistics while still being able to show more specific details on single flows upon request and proper algorithmic reversal.

In this section we propose a general purpose and flexible counting framework that addresses the previously discussed requirements, and that can be used in a broad plethora of network applications. The adopted approach is that of "trading certainty for time/space" [110] by using probabilistic algorithms at the cost of a small and tolerable error rate. In a nutshell, the proposed counting framework combines an efficient probabilistic counter (the *LogLog counter* [43] already shown in chapter 1) to a compact and privacy-preserving probabilistic data structures sketches, acting as containers, and that can be "reversed" only upon specific conditions are met. Overall, the whole data structure inherits the low complexity and memory efficiency of probabilistic counters and Bloom filter like structures, while proving insensitivity to data duplication and allowing the identification of the source of investigated data. Beside the architectural view of the counting framework, this section presents a practical and efficient implementation of the proposed data structure, also made available to the community at the address **https://github.com/awgn/pds** for free downloading.

This work unifies the previous research presented in [29] and [30] and extends their finding by refining the overall architecture, by integrating a more efficient counter and by proposing a real high-performing C++ implementation of the whole counting data-structure.

5.3.1 Two motivating use-cases

This section presents two real-world use-cases in which network monitoring and security applications, respectively, require the use of efficient data structures for high-speed data processing. As it will be elaborated upon, the first use-case is induced by the recently emerged Software Defined Networking (SDN) paradigm and the new way of thinking of network applications themselves according to this philosophy. The second use-case, instead, refers to a more traditional architecture of anomaly detection through distributed measurement points over backbone networks. In both cases, it will be evident that performance is not the only addressed requirement as the management of data collected from multiple vantage points and privacy preservation are not less important issues to handle.

SDN and Internet eXchange Points monitoring

According to the Open Networking Foundation [4] a Software Defined Network is defined as an architecture that decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. Hence, the key point in such networks is the clear distinction between the control plane and the data plane. Indeed, while in a “classical" network, the two co-exist in a single apparatus (e.g., the router), in an SDN the control and the data plane are managed by distinct devices.

From an architectural point of view (as shown in Fig. 5.12), SDN is composed of three distinct layers. The lowest layer, namely the infrastructure layer, represents the data plane of the network and it is composed of either real or virtual device, named switches. Their function is to forward a packet by some rules.

The intermediate level is the control layer, sometimes referred to as Network Operating System or, just, controller. It is responsible – on one side – to manage the switches and inject the rules, and – on the other – to provide the upper layer with an abstract view of the underlying network.

Finally, the top layer, named application layer, is composed of business applications able to perform a wide range of actions (e.g., path computation, firewall, monitoring).

The communication among the different layers is permitted by the *southbound* APIs, between the infrastructure layer and the controller, and by the *northbound* APIs, between the controller and the applications. Regarding the southbound API, the most used protocol is OpenFlow [5] [73], and a device managed through the OpenFlow protocol is named OpenFlow Switch (simply referred to as switch in the rest of the section). This kind of switch consists of a flow table (or N flow tables) that specifies the actions

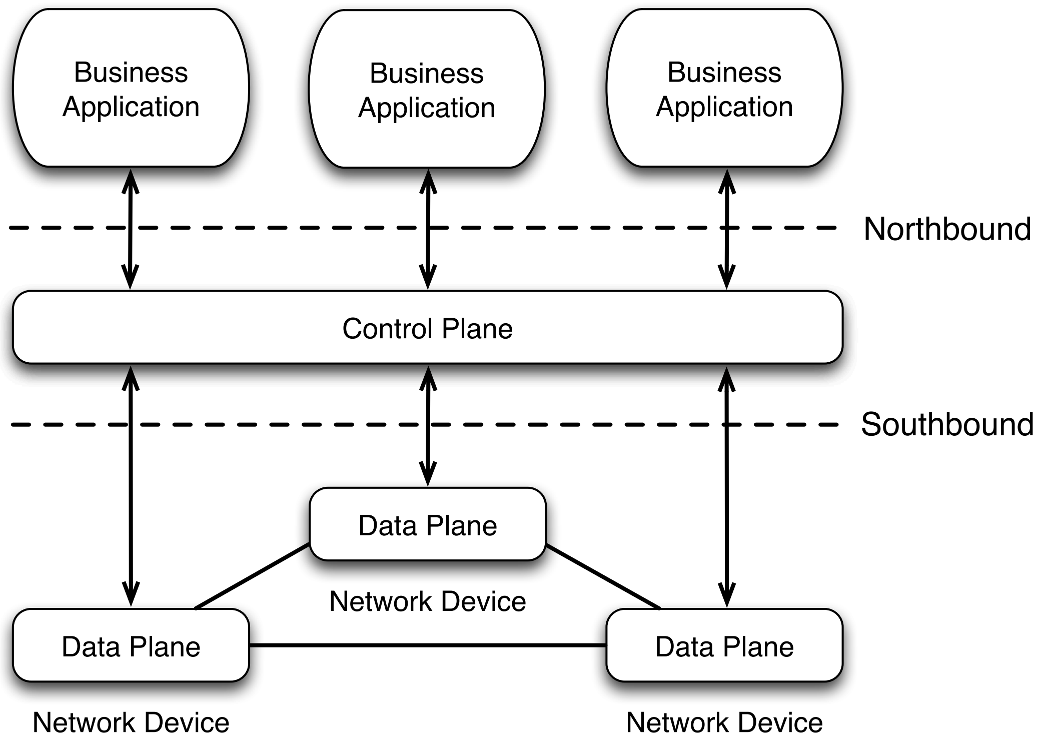


Figure 5.12: *SDN Architecture*

to be executed on each received packet and a secure channel to communicate with the controller.

In more detail, a flow table consists of several entries, each composed of *fields* that match some packet header fields, *counters* updated every time such *fields* are matched, and *actions* to be applied to the matched packets. Such tables can be inserted/modified/deleted by the controller only, which is also able to require some data (e.g., counter values) from the switch.

Finally, the northbound APIs are used from the application layer to ask the controller data/information that can be used to perform several tasks (e.g., traffic monitoring, anomaly detection).

From the functional point of view, upon receiving a packet from the network, a switch works in the following way (see also Fig. 5.13): first, it parses the packet headers to check for possible matches between these and the *fields* specified in any of the rules stored in the different flow tables. If this is the case, the corresponding *actions* (e.g., forward) are undertaken, otherwise the packet is encapsulated into a new packet and sent to the controller through a secure channel. Hence, the controller will create a rule for such a packet and will update the switch flow tables accordingly.

It is important to highlight that the number of “unknown" packets (i.e., packets for which the switch has not a rule that specifies an action) should be low during “normal" network behavior, for not overwhelming the controller. In any case, such packets can be considered as representative indications of some anomalous network behavior.

On the Internet, the different “independent" networks are connected through what

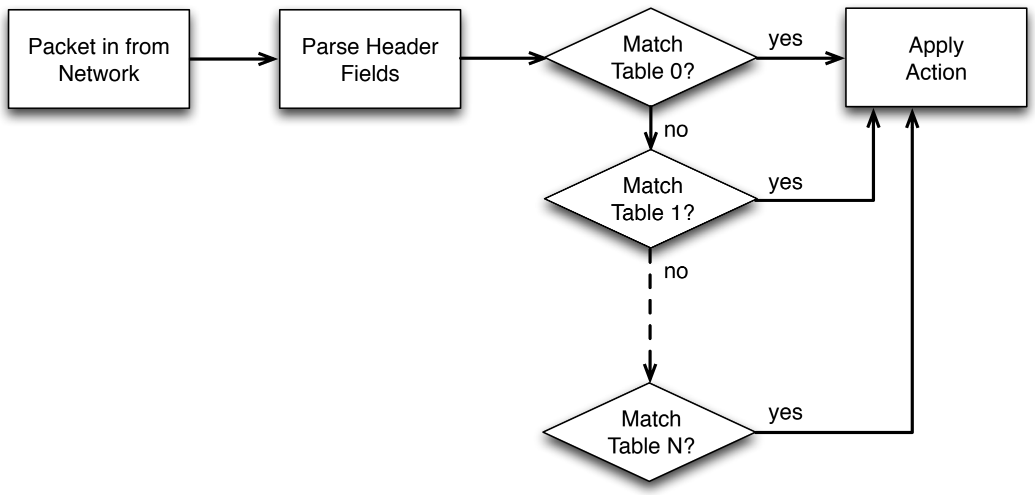


Figure 5.13: *Packet Processing in the switch*

is known as peer relationships. Hence, packets are forwarded towards the destination, passing from one network to another, because such networks have a peering relationship with each other. The point where the different networks interconnect and such peering relationships are established is named Internet Exchanges Point (IXP).

In a nutshell, at its most basic level, an IXP is a switch into which multiple networks connect and can then pass bandwidth. More realistically, an IXP is a Layer 2 network.

Given such a definition, it is clear that IXPs represent a “natural" deployment field for SDN technologies, (as an example, see SDX [54] and CARDIGAN Project [104]).

Such a scenario represents a good use case for our proposed architecture. Indeed, it is well known that the security problems in an IXP [61] are usually due to:

- Unhygienic routers
- BGP manipulation
- Network capacity theft

While the first two issues can be solved (or, at least, significantly alleviated) using a meticulous configuration of the IXP apparatus, the network capacity theft is mainly due to processing and forwarding of traffic flows that are not supposed to get through the IXP. Hence, it is clear that counting and identifying such “unknown" flows is of primary interest in such a context. In the following, we will show that thr proposed architecture proves to be well tailored to this case by providing the details of the algorithms as well as the results obtained from the experimental analysis.

Anomaly detection in Backbone Networks

In the last few years, Internet has experienced explosive growth. Along with the wide proliferation of new services, the quantity and impact of attacks have been continuously increasing. The number of computer systems and their vulnerabilities has been rising, while the level of sophistication and knowledge required to carry out an attack have been decreasing, as much technical attack know-how is readily available on Web sites all over the world.

As a consequence, many research groups have focused their attention on developing novel detection techniques, able to promptly reveal and identify network attacks, mainly detecting Heavy Changes (HCs) in the traffic volume [14], [27], [67], [114], and [109]. Nevertheless, the recent spread of coordinated attacks, such as large-scale stealthy scans, worm outbreaks, and distributed denial-of-service (DDoS) attacks that occur in multiple networks simultaneously, makes challenging the detection by using isolated intrusion detection systems that only monitor a limited portion of the Internet. Hence, the research efforts are now moving to develop distributed approaches to solving such an issue [115].

In distributed anomaly detection algorithms, multiple detection probes – distributed in the backbone network – monitor a given portion of the network separately and report the collected information to a single location (named *mediator*) that analyzes the data and generates the alerts. By limiting the scope to the simplest case of anomaly detection algorithms that analyze traffic volumes only, the data collected by the probes are represented by the estimation of the number of traffic flows observed in a given time window. Hence, the first problem to be solved is to provide a reliable estimate of such quantities. This task, that is not trivial when performed over the multi-gigabits links of a backbone network, has been discussed in several previous works, and the use of probabilistic data structure has emerged as a *standard* approach [49].

This task is performed by the distributed probes, that then forward the estimated data to the mediator that is responsible for aggregating them. It is worth noticing here that, when aggregating these structures at the mediator level, the problem of *not counting* duplicated flows – i.e., the flows observed by more than a single probe – must be solved.

In general, the counters values associated with specific flows represent sensitive users information and must not be openly disclosed. The use of probabilistic data structures like sketches [49] provides an ideal container to keep all such data in an aggregated way. However, whenever traffic anomalies are detected, a method of identifying the flows (in particular, IP addresses) responsible for the supposed misbehavior is needed. As it will be shown, the counting framework hereafter presented solves this issues by allowing a controlled reversal of suspicious data only, with no impact on regular ("normal") traffic.

5.3.2 The probabilistic counting framework

As above discussed, the counting framework proposed is "doubly probabilistic" in that it comes from the combination of a probabilistic data structure and a probabilistic counter. At the high level, the data structure is a three-dimensional array $S_{D\times W\times L}$, where each row d ($d = 1, \ldots, D$) is associated with a function δ_d that takes values in the interval $(1, \ldots, W)$ that are associated with the columns of the array.

Note that the two-dimensional substructure $S_{D\times W}$ is a *standard* reversible sketch table. The third dimension L is introduced to integrate the LogLog counting algorithm in such a sketch table. To this aim, each bucket of the sketch is associated with another hash function H_{dw} that gives output in the interval $(1, \ldots, L)$, associated with the layer (depth) of the array. In the construction phase, we have chosen to use hash functions

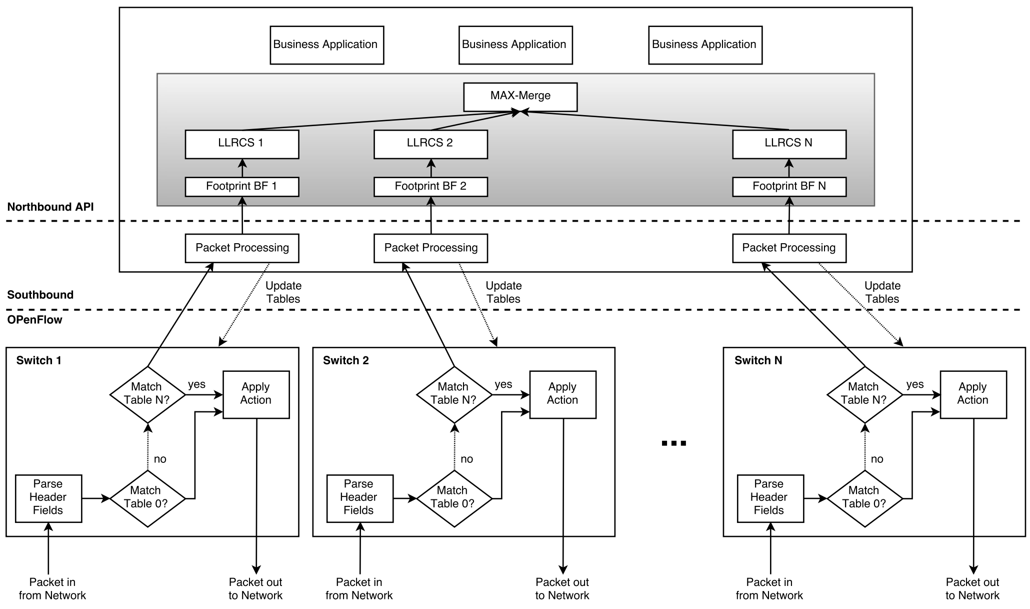


Figure 5.14: *SDN-IXP Counter of Different Flows*

that belong to the 4-universal hash family[4] [108], obtained as:

$$h(x) = \sum_{i=0}^{3} a_i \cdot x^i \mod p \mod W \tag{5.1}$$

where the coefficients a_i are arbitrarily chosen in the set $\{0, 1, \ldots, p-1\}$ and p is a random prime number (we used the Mersenne ones). Updating the data structure require first to choose the key of interest and apply mangling and hashing functions to real data (to be counted) to select a bucket $S[d][w][\cdot]$ and the associated hash function H_{dw}. For all the keys that collide in a given bucket, the system computes the flow ID, typically given by a function of the header fields IP source and destination addresses, source and destination ports, and protocol. Then, it computes the hash function H_{dw} of the flow ID and updates the LogLog counter accordingly.

Depending on the specific applications – this is the case of both the network applications addressed in this section – there might need to combine several data structures as a result of multiple instances of traffic measurements. More formally, each counter of the sketch represents the cardinality of the support of a multiset. When measurements are obtained at different points, and the results combined, each bucket ends up

[4]A class of hash functions $H : (1, \ldots, N) \to (1, \ldots, W)$ is a *k-universal hash* if for any distinct $x_0, \cdots x_{k-1} \in (1, \ldots, N)$ and any possible $v_0, \cdots v_{k-1} \in (1, \ldots, W)$:

$$Pr_{h \in H}\{h(x_i) = v_i; \forall i \in (1, \ldots, k)\} = \frac{1}{W^k}$$

representing the sum of multisets (namely, a multiset that accommodates the items of each multiset together with the sum of their multiplicity) whose support is the union of the supports of all multisets. It is easy to convince ourselves that merging the data structures by means of the max-merge algorithm:

$$M[d][w][l] = \max_{p} S^p[d][w][l] \tag{5.2}$$

gives an estimation for the cardinality of the support set, that is the number of distinct elements of the aggregate multiset. Obviously, to allow this operation, all the different sketches of counters must have been constructed using the same hash functions.

Depending on the applications, a parallel data structure representing a reversible sketch (RS) [95] may be constructed at each measurements point for identification purposes. Once again, RS must share common hash functions to apply the reverse algorithm.

Counting Unknown Flows in SDN-Based IXPs

The first application of the general purpose probabilistic counters described in the previous section refers to the SDN-IXP use-case. More precisely, the problem is to realize a monitoring application that estimates the number of different flows in an SDN network. Our application is developed as a business application on top of the SDN controller. The overall scheme of the network application is shown in Figure 5.14.

After receiving a packet for which there is no corresponding flow table entry, the switch sends such a packet to the controller, through the OpenFlow protocol.

Hence, the controller first processes the packet and sets a new entry in the switch flow table by using OpenFlow, then passes the packet to the counter application that inserts the related flow key in a footprint Bloom filter and then updates the LogLog Counting Reversible Sketch table (LLRCS) – the reason why the controller must keep a distinct table for each switch will be clear in the following.

Once all the LLRCSs corresponding to the different switches have been constructed, they are merged through (5.2).

At this point, by applying the equation (1.11) to each bucket $M[d][w][\cdot]$, the controller has an estimate of the “unknown” flows that collide in the same bucket. If needed (e.g., in case they exceed some given threshold), it can identify the flows by applying the reversible sketch algorithm and checking its output over the footprint Bloom filter.

Notice that such an apparently complex architecture solves the problem of counting the “unknown” flows in an SDN network, and is needed for several reasons, as discussed in the following points.

- The potentially high number of unknown flows in a network makes the use of deterministic data structure unfeasible; hence the sketch family data structure represents an optimal choice for such a kind of a problem.
- The use of a *standard* sketch for counting the flows (e.g., a count-min sketch [36]) is not satisfactory in our scenario, since they do not offer a means to automatically discard duplicates (i.e., the same flow traversing more switches).
- The use of a reversible sketch is justified by the fact that, once the unknown flows are identified, the OpenFlow protocol allows to modify the behavior of the switch traversed by such flows.

- The choice of computing a distinct LLCRS for each switch is needed to isolate the network segment in which such flows are observed. Note that, in case this is not strictly necessary, the controller can simply compute a single LLCRS, skipping the max-merge phase.

It is important to highlight that in several domains (and quite likely in the IXP/SDN scenarios), to solve the scalability issues typical of SDN networks, more than a single controller is used – hierarchical architecture. Our solution can easily be adapted to such a scenario, by observing that the LLCRS computed by a controller can be sent to higher hierarchical levels for further aggregations.

Probabilistic Counting for Anomaly Detection in Backbone Networks

This section presents the application of the probabilistic counting framework to the anomaly detection use-case by showing the scheme of a distributed and collaborative Intrusion Detection system. The overall architecture of the application is shown in Figure 5.15

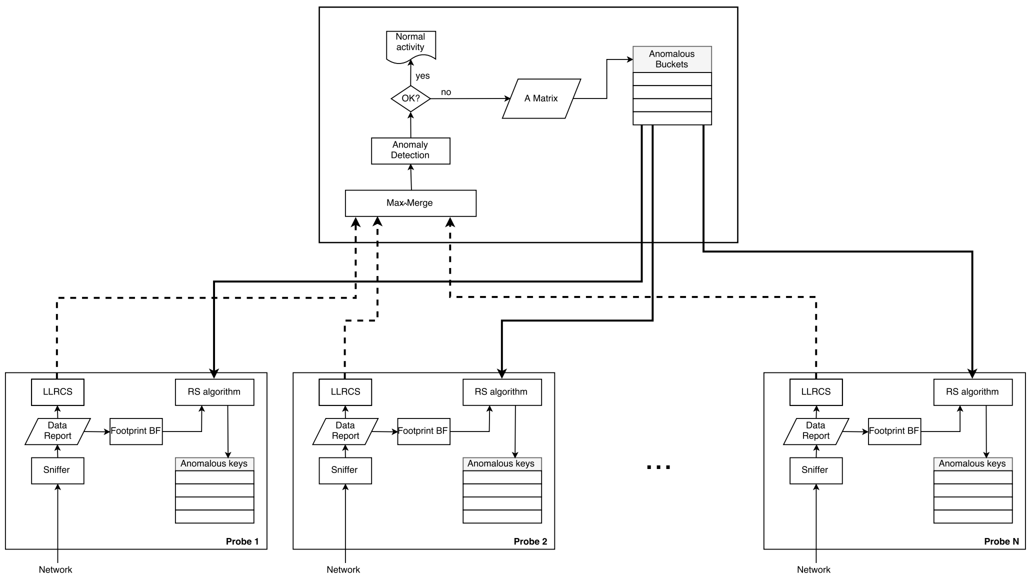


Figure 5.15: *Anomaly Detection Application*

Multiple detection probes – built on-top-of PFQ framework – are distributed in the network to monitor separate segments and report the collected information to a single location (mediator) in charge of processing data and raise alerts for suspicious traffic activity. A backtrack mechanism is then used by the mediator to ask the probes for flow identification.

Starting from the observed traffic, each probe produces a periodic report that contains information related to the traffic measured in a given time bin, which consists of a collection of flow keys: $\langle$*IP source address, IP destination address, source Port, destination Port, Protocol*$\rangle$.

The periodic reports are passed to the module responsible for the construction and the update of the LogLog Counting Reversible Sketch, as well as the footprint Bloom filter. At this stage, each probe has a summary of all the different observed flows, together with an estimate of their number.

The mediator, responsible for the detection phase, combines the information exported by each probe through the equation (5.2).

Since the max-merge operation implicitly solves the problem of not counting duplicated flows, the resulting aggregated sketch is precisely equivalent to the one that would be constructed in an "ideal" case, with the mediator directly observing the whole traffic. This properity also implies that the estimation error, due to the probabilistic nature of the data structure, is not worsened by the distributed nature of the application, being equivalent to that of a single probabilistic counter.

At this point, the mediator counts the number of distinct flows that collide in the same bucket. This task can be solved by applying equation (1.11) to each bucket $M[d][w][\cdot]$.

By using a classical detection method (e.g., PCA, wavelet analysis, heavy hitter) the system can decide whether there are or are not anomalous aggregates in a given time bin. The output of this phase is a binary matrix ($A[d][w]$) that contains a "1" if the corresponding bucket is considered anomalous "0", otherwise.

Note that, given the nature of the sketches, each traffic flow is part of several random aggregates (namely D aggregates), corresponding to the D different hash functions. This means that, in practice, any flow will be checked D times to verify if it presents an anomaly (this is done because an anomalous flow could be masked in a given traffic aggregate, while being detectable in another one).

Due to this fact, a voting algorithm is applied to the matrix A. The algorithm verifies if at least H rows of A contain at least a bucket set to "1" (H is a tunable parameter). If so the mediator reveals an anomaly, otherwise, the matrix A is discarded.

In case the mediator reveals some anomalous time bin during the detection phase, it back-propagates the matrix $A[d][w]$ to the probes.

At this point, each probe uses the RS computed for the anomalous time bin and the footprint Bloom filter to identify the IP addresses responsible for the detected anomalies.

5.3.3 C++ implementation

The probabilistic framework has been implemented in C++ within the `pds` (Probabilistic Data Structures) library and is freely available for download at [7]. The library includes the implementation of some randomized data structures, such as Bloom filters, Counting Bloom filters, Sketches, LogLog counters (and their variants), as well as a set of utility functions, algorithms, and tests.

The library is designed according to the following principles:

- *Generic Programming and Composability*. Overall, the library pervasively uses templates to ease code reusability. All data structures are indeed abstract and implemented as containers of generic data type that can be specialized upon need.

As an example, the sketch buckets may contain any data (e.g., integer, arrays, LogLog counters, etc.) equipped with proper operations.

- *Robustness*. The inherent complexity of managing advanced data structures may easily lead to configuration errors (e.g., types of data, hash functions codomain bitwise length, etc.). The use of template meta-programming techniques allows to spot such errors at compile-time and enforce correctness and consistency of the declared parameters using static asserts.
- *Declarative syntax*. The properties of all data structures are embedded within their declaration, as it improves the usability of the library. As an example, a sketch declaration consists of the size of the sketch itself, the type of data accommodated in the buckets, the hash functions used in each row together with the codomain bitwise dimensions.

The following subsections present more details about the implementation of reversible sketches and LogLog counters as they are used this work.

LogLog counters

The use of a LogLog counter `ll` is simply instantiated as follows:

```
using LogLog_type =
      pds::LogLog< uint8_t
                 , 1024
                 , std::hash<std::string>>;
LogLog_type ll;
```

The statement declares a LogLog counter suitable for estimating the cardinality of a multiset of strings. The number of "small bytes" is set to 1024 while their type is an unsigned byte (although 5 bits are sufficient). In the example, the strings are hashed through the standard hash function for strings, though any other function type can be used as parameter of the template declaration.

Hyper LogLog counters are also implemented in the library and their use only requires replacing the name `HyperLogLog_type` to `LogLog_type` in the above declaration.

Sketches

An example of sketch declaration for string counting is the following:

```
using sketch_type =
    pds::sketch< uint32_t
               , 1024
               , BIT_10(myhash1<string>)
               , BIT_10(myhash2<string>)>;

sketch_type s;
```

The `using` statement declares a sketch of 1024 columns and two rows, each with a different user-defined hash function. The `BIT_10` macros are herein used to annotate that the co-domain bit-size of such hash functions is set to 10. At compile-time, suitable meta-functions evaluate the hash sizes and verify the consistency with the sketch size.

The class provides a broad number of iterators that allows visiting the buckets (one per line) given a specific element to be inserted in the sketch. For instance, the method

```
s.foreach_bucket("fortytwo", [](int &bkt) {
    bkt++;
});
```

iterates over the all the buckets associated with the string `fortytwo` and updates the content using the given lambda function. For standard operations like increment or decrement, methods like increment_buckets and decrement_buckets are also provided.

Besides, the sketch class provides the count-min estimation, the k-ary estimation, and a set of more general functions for the sketch management, including filtering buckets, searching elements and sketches aggregation.

Reversible Sketches

The reverse sketch algorithm [95] requires the bitwise partitioning of the keys (elements) used to update the sketch and that each part of the keys (word) is hashed separately. In the library abstraction, elements are represented as *tuples* of generic types in which each component of the tuple represents a portion of the original key. The modular hash is then applied to the tuple as a whole by concatenating the hash values of each component of the tuple. This representation is particularly convenient in networking use cases as standard keys come from the concatenation of different packet header fields, such as IP addresses, TCP ports, protocol field, and so on. Notice that, even single field keys (e.g., IP addresses), can be conveniently split into several parts to improve the efficiency of the algorithm.

As an example, the C++ key for a TCP flow can be represented as:

```
auto key = std::make_tuple( ip_src.high
                          , ip_src.low
                          , ip_dst.high
                          , ip_dst.low
                          , src_port
                          , dst_port
                          , proto);
```

where source and destination IP addresses are split into two 16-bits long fields.

The library provides the implementation of a modular hash for generic N component tuples that is obtained by composing N different hash subfunctions. Each subfunction is applied to a single component of the tuple; the final result is obtained by concatenating the output of all subfunctions. Notice that both the bitwise length of the tuple components, as well as the bitwise length of the output of the hash subfunction is fully configurable at compile-time.

For example, the following declaration defines a modular hash function for the above-presented TCP flow tuple.

```
using hash_type = pds::ModularHash< BIT_4(H1)
                                  , BIT_4(H1)
                                  , BIT_4(H1)
                                  , BIT_4(H1)
                                  , BIT_3(H2)
                                  , BIT_3(H2)
                                  , BIT_3(H3) >
```

It is worth noticing that three different functions `H1`, `H2`, and `H3` are used (but a single function could have been used as well). The output of `H1` is constrained to be 4 bits long, while H2 and H3 produce 3 bits long results each. Overall, the complete hash functions yields a 25 bits long output and is obviously represented as a 64 bit integer.

Therefore, a reversible sketch s that uses the canonical IP flow can be defined as follows (for the sake of simplicity, the same hash function is applied to all the components of the tuple):

```
pds::sketch< uint16_t
           , (1<< 21)
           , pds::ModularHash< BIT_4(H1)
                             , BIT_4(H1)
                             , BIT_4(H1)
                             , BIT_4(H1)
                             , BIT_3(H1)
                             , BIT_3(H1)
                             , BIT_3(H1) >
           , pds::ModularHash< BIT_4(H2)
                             , BIT_4(H2)
                             , BIT_4(H2)
                             , BIT_4(H2)
                             , BIT_3(H2)
                             , BIT_3(H2)
                             , BIT_3(H2) >
           , pds::ModularHash< BIT_4(H3)
                             , BIT_4(H3)
                             , BIT_4(H3)
                             , BIT_4(H3)
                             , BIT_3(H3)
                             , BIT_3(H3)
                             , BIT_3(H3) >
           , pds::ModularHash< BIT_4(H4)
                             , BIT_4(H4)
                             , BIT_4(H4)
                             , BIT_4(H4)
                             , BIT_3(H4)
                             , BIT_3(H4)
                             , BIT_3(H4) >
           , pds::ModularHash< BIT_4(H5)
                             , BIT_4(H5)
                             , BIT_4(H5)
                             , BIT_4(H5)
                             , BIT_3(H5)
                             , BIT_3(H5)
                             , BIT_3(H5) >
           , pds::ModularHash< BIT_4(H6)
                             , BIT_4(H6)
                             , BIT_4(H6)
                             , BIT_4(H6)
                             , BIT_3(H6)
                             , BIT_3(H6)
                             , BIT_3(H6) >
       > s;
```

Sketch updates are made through the increment_buckets method. The following code emulates the TCP flow

$$< 0\text{x}bad, 0\text{x}bee, 0\text{x}dead, 0\text{x}beef, 6010, 4216, 6 >$$

and the UDP flow

$$< 0\text{x}dead, 0\text{x}beef, 0\text{x}cafe, 0\text{x}babe, 80, 6667, 17 >$$

both hitting the sketch 1000 times and triggering the increment of the associated buckets, accordingly.

```
for(auto n = 0; n < 1000; n++)
{
    s.increment_buckets(std::make_tuple(
    0xbad, 0xbee, 0xdead, 0xbeef, 6010, 4216, 6)
                        );
    s.increment_buckets(std::make_tuple(
    0xdead, 0xbeef, 0xcafe, 0xbabe, 80, 6667, 17)
                        );
}
```

The method `index_buckets` is used to compute a predicate (in the example the passed lambda function) over the whole sketch. In the following example, the method is used to retrieve the buckets whose content exceeds 500. The result is a matrix (vector of vectors) that contains the indexes of the buckets that satisfy the predicate.

```
    auto idx = s.index_buckets([](auto &b)
    {
        return b > 500;
    });
```

The final step is to feed the method `reverse_sketch` with the above-obtained bucket indexes to obtain the list of the tuples (i.e., keys) that hit the sketch in the selected buckets. The following code is used to reverse the sketch.

```
auto rev = pds::reverse_sketch< uint16_t
                              , uint16_t
                              , uint16_t
                              , uint16_t
                              , uint16_t
                              , uint16_t
                              , uint8_t >(s, idx);
```

LogLog Counting Reversible Sketches

The LogLog counting reversible sketch is obtained by wrapping up the previously described components and defining each bucket of the sketch to be a LogLog counter.

The following example refers to the simple case of a *port scan detector*. In this scenario, the attacker is an Internet host that sends SYN packets with different destination port numbers to check for existing TCP services.

The sketch is then populated by hashing on the pair source and destination IP addresses that, in the example, are both conveniently split into two parts. The LogLog counter, instead, is updated by using the pair source/destination ports and will only get incremented upon ports variation.

```
using LogLog_t =
    pds::hyperLogLog< uint8_t
                    , 64
                    ,  std::hash<std::tuple<uint16_t, uint16_t>>
                    >;
pds::sketch< LogLog_t
           , (1 << 16)
           , pds::ModularHash< BIT_4(H1)
                             , BIT_4(H1)
                             , BIT_4(H1)
                             , BIT_4(H1) >
           , pds::ModularHash< BIT_4(H2)
                             , BIT_4(H2)
                             , BIT_4(H2)
```

```
                                    , BIT_4(H2) >
              , pds::ModularHash< BIT_4(H3)
                                    , BIT_4(H3)
                                    , BIT_4(H3)
                                    , BIT_4(H3) >
              , pds::ModularHash< BIT_4(H4)
                                    , BIT_4(H4)
                                    , BIT_4(H4)
                                    , BIT_4(H4) >
              , pds::ModularHash< BIT_4(H5)
                                    , BIT_4(H5)
                                    , BIT_4(H5)
                                    , BIT_4(H5) >
              > s;
```

The method `cardinality` is used to retrieve the index of the LogLog counters with values bigger than 10000.

```
auto idx = s.index_buckets([](auto &b)
{
    return b.cardinality() > 10000;
});
```

Likewise, the list of candidates is obtained by reverting the sketch through the following code:

```
auto rev = pds::reverse_sketch< uint16_t
                               , uint16_t
                               , uint16_t
                               , uint16_t
                               > (s,idx);
```

5.3.4 Experimental results

The doubly probabilistic nature of the overall counting framework introduces two possible levels of errors whose effects are not always easily predictable in practice. The first obvious source of uncertainty is given by the statistic nature of the counter for which, however, theoretical bounds are available. The second, and the more subtle, source of errors is instead given by the collision that may occur when populating the sketch. Indeed, if erroneously detected keys can be readily discarded by a simple footprint Bloom filter, the effect of collisions in a counter of the sketch cannot be depurated and requires the sketch size to be thoroughly configured according to the application requirements to avoid measurement corruption.

In this section we present the performance assessment of the proposed system in the pretty general case of the *heavy hitter* detector application. The application itself can run on a single vantage point or in a distributed fashion and is a particular case of the anomaly detection algorithm presented in section 5.3.2. Upon receiving a packet, the probes involved in the measurement extract the source IP address and use it as the key to increment the corresponding buckets of its LLRCS sketch. Each probe periodically sends the LLRCS up to the mediator that merges the received LLRCSs and check if the traffic volume recorded in some buckets exceeds a given threshold (typically expressed as a percentage of the total recorded traffic load). In the affirmative case, the mediator sends the address of the “anomalous” bucket back to the probes which, in turn, run the reverse algorithm and refine the results by the footprint Bloom filter to come up with a list of responsible *heavy hitters* (IP candidates).

Although quite simple, such a use case involves a complete set of critical parameters that need to be investigated to assess the performance of the whole system. More in details, the following performance indexes will be investigated:

- Sketch size
- False alarm rate

All of the two parameters are strictly correlated and point directly to the primary motivation of this work. Indeed, we advocated the use of our system because of the compactness of the data structures that, in turn, facilitates their transfer and storage in small memory devices. However, the size of the data structure is determined by two factors: i) the size of the sketch (namely, the number of row, 4 in our tests, and columns, determined by the length of the hash functions) and ii) the depth of the counter (i.e., the size and the number of the small bytes). Obviously, the smaller the data structure, the higher the false positive rate (due to the more substantial number of collisions) and the lower the estimation accuracy (due to the smaller size of the counters).

In all experiments, the application run in stand-alone mode as the distributed behavior would not add any valuable insights to the performance analysis. Hence, a single probe was fed with the real trace from the MAWI repository [53], containing 928223 distinct flows and 260851 distinct IP source addresses, corresponding to time 14:00 of September 28, 2016. Also, a trace containing a synthetic heavy hitter was mixed in case the original trace would not include any high hitting flow to detect.

Starting with the analysis of the sketch size, in Table 5.1 we present the compression factor achieved by the sketch in comparison with a C++ unordered map (containing the deterministic and exact counters). We can easily see that the compression rate decrease when increasing the hash output length (number of columns) or the LogLog bucket size. Such results are not significant by themselves, since as already stated in the previous sections the bigger the sketch, the better the performance of the counters, and must be commented together with the following ones.

Hence, moving to the False Alarm rate, it is worth highlighting that it depends on three distinct components:

- post-filtering phase performed by means of the *footprint* Bloom filter (as explained in Section 1.6.2)
- number of collisions
- Count estimation error

Regarding the first one, Tables 5.2 and 5.3 respectively show the performance achieved with and without the use of the *footprint* Bloom filter. In our experiments the BF length m has been set according to the optimal size of $m = n*k/\log(2)$, where k is the number of hash functions (4 in our case), and n is the expected number of flows, rounded up to the smallest power of 2 (512kByte in our tests).

It is easy to understand that the post-filtering phase only affects the performance in case of "short" hash functions and is negligible in all the other cases. Indeed, the performance is significantly improved for all of the cases corresponding to 8 and 10. It is worth noticing that in the cases 8, the use of the BFs is of primary importance. Indeed, without such a filter, not only would the system reveal almost all of the observed flows

Table 5.1: *Compression Factor*

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	603.0976	301.5488	150.7744	75.3872
10	150.7744	75.3872	37.6936	18.8468
12	37.6936	18.8468	9.4234	4.7117
14	9.4234	4.7117	2.35585	1.177925
16	2.35585	1.177925	0.5889625	0.29448125

Table 5.2: *False Alarm Rate (%)*

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	65.38	66.38	66.3777	67.09
10	0.03	0.023	0.017	0.013
12	0.0023	0.0027	0.0034	0.0031
14	0.0019	0.00157	0.0031	0.0027
16	0.0016	0.00157	0.0019	0.0027

as potential candidates, but it would also indicate a large number of unobserved flows as potential candidates (this is due to the nature of the RS algorithm).

Moving to the impact of the collisions and the estimation error on the false alarm rate, it is essential to specify that it is not possible to precisely analyze their impact, separately. Nonetheless, it is very intuitive that collisions depend on the hash length, while the estimation error depends on the bucket size, as clearly demonstrated by the results shown in Table 5.2. To better evaluate the Count estimation error, in Tables 5.4, 5.5, and 5.6 we show the percentage of flows that are in within $\pm\sigma$, $\pm 2\sigma$ and $\pm 3\sigma$ of the exact count. As expected we can see that in all of the cases the constraints (described in Section 1.6.1 of the chapter 1) are met. Such results are also visually depicted in Figure 5.16 5.17 and 5.18, where we show three reasonable cases (as it will be later discussed).

Summing up, by inspecting the different performance figures, it is possible to select a set of cases that offer the best trade-off between memory footprint and detection probability. In general, all of the cases corresponding to hash length of 10 and 12 bits provide acceptable performance and the choice among them can be driven by memory availability. Indeed, taking as an example the case with the hash length of 10 bits

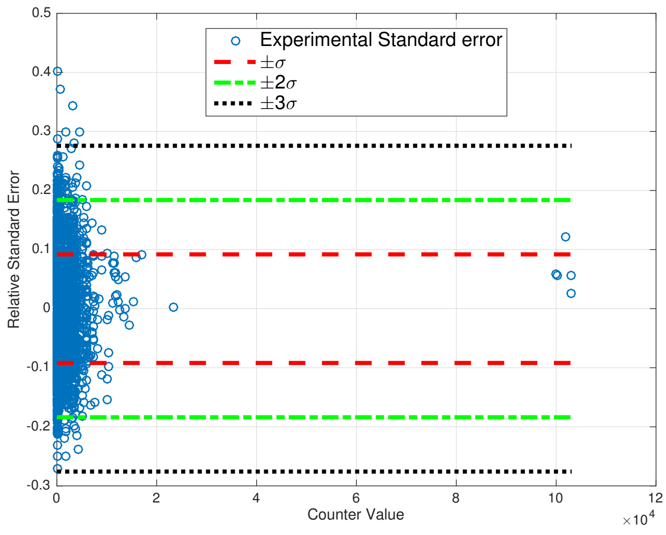


Figure 5.16: *Count error estimation: Hash Length = 10 – Bucket Size = 128*

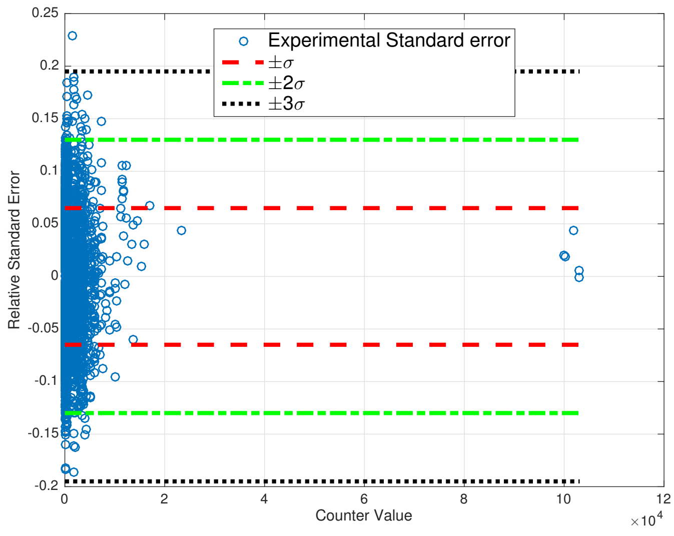


Figure 5.17: *Count error estimation: Hash Length = 10 – Bucket Size = 256*

Table 5.3: *False Alarm Rate (%) – without footprint BF*

	LogLog Bucket Size			
Hash Length	32	64	128	256
8	99.99994	99.99994	99.99994	99.99994
10	0.16369	0.12267	0.07629	0.06670
12	0.00230	0.00268	0.00345	0.00306
14	0.00191	0.00153	0.00306	0.00268
16	0.00153	0.00153	0.00191	0.00268

Table 5.4: *Count Estimation within* $\pm\sigma$

	LogLog Bucket Size			
Hash Length	32	64	128	256
8	0.721875	0.726562	0.735156	0.746094
10	0.726758	0.749805	0.777734	0.805469
12	0.811667	0.827997	0.847474	0.857016
14	0.889683	0.906507	0.924965	0.934547
16	0.92347	0.937113	0.949509	0.949509

and bucket size of 128 bits, the system presents a false alarm rate of 0.017% with a compression factor of around 37.7, which make the data structure rather thin while still providing excellent detection performance.

Table 5.5: *Count Estimation within* $\pm 2\sigma$

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	0.957812	0.9625	0.964063	0.975
10	0.964063	0.97207	0.981055	0.988477
12	0.981555	0.984064	0.981063	0.977915
14	0.986909	0.986333	0.983464	0.983577
16	0.989282	0.988879	0.987806	0.988549

Table 5.6: *Count Estimation within* $\pm 3\sigma$

Hash Length	LogLog Bucket Size			
	32	64	128	256
8	0.989062	0.99375	0.996875	0.999219
10	0.993359	0.996289	0.998633	0.999805
12	0.998033	0.998574	0.998229	0.995672
14	0.999327	0.99686	0.995834	0.993767
16	0.999377	0.996993	0.996048	0.995141

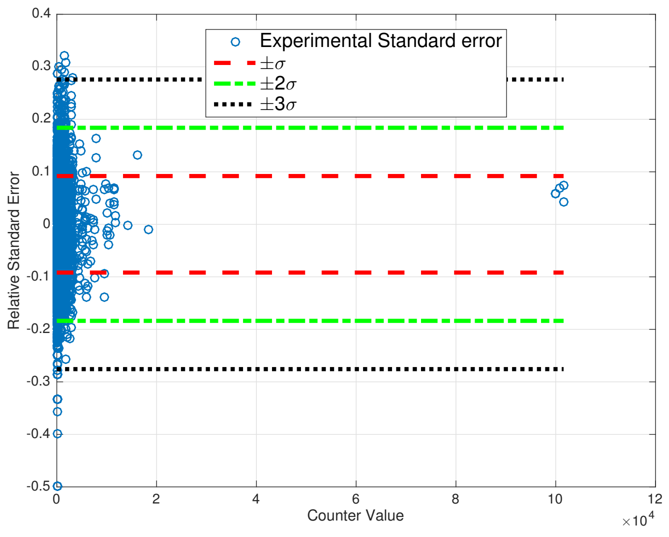


Figure 5.18: *Count error estimation: Hash Length = 12 – Bucket Size = 128*

CHAPTER 6

Conclusions

The continuous growth of internet traffic, coupled with the ever-increasing number of applications and the speeds of links, calls for the need of purely software solutions that must be flexible, re-configurable and high-performing by design. As a consequence, applications in charge of processing such a vast amount of data are becoming more and more complicated.

The network-oriented software can no longer be developed with the techniques adopted since the 90's but still in use today, and require the use of sophisticated semantics to take advantage of the parallelism of modern hardware (e.g., multi-threading, thread pinning, interrupt affinity, to mention a few).

This thesis aims at analyzing all layers of which a network application is composed, starting from the lowest one, the network device driver, up to the highest layer, where the processing of network packets takes place. Our primary goal is to find all the obstacles and bottlenecks that can affect the performance of applications designed to work in multi-gigabit environments. Hence, the thesis presents a framework designed to bypass the inefficiencies of standard applications that provide essential accelerations without requiring source code intervention.

In the first chapter, we analyze some bottlenecks introduced by well-established programming techniques applied to multi-processor architectures. We recognize multi-process and multi-thread programming as a viable and necessary condition for the exploitation of multi-core architectures. With simple examples, however, we show how some of these methodologies are no longer suitable in the context of network programming, where speed is one of the primary objectives.

We thus set the key elements to handle shared data structures, without using spin-lock or mutex to prevent date race conditions. Atomic operations, such as the exchange or the compare and swap (CAS), become fundamental building-blocks for creating lock-free or wait-free algorithms.

Besides, we take a look at the use of memory by introducing the true- and false-sharing concept, providing a strategy for boosting performance with an example of a distributed counter. Finally, we present a comparison of several implementations of an SPSC queue, showing how performance can be improved by using a cache-aware memory allocation and adopting a soft-cache to undermine the issue of lines' coherence in multi-core architectures.

In the second chapter, we describe PFQ, an open source network processing framework for the Linux OS designed to provide a flexible and powerful platform for the development of parallel network applications. At the lower level, PFQ implements a set of software accelerated techniques to efficiently handle traffic captured and transmitted over standard network device drivers. At a higher level, the platform integrates an in-kernel programmable engine to perform early-stage processing and custom-defined distribution to user-space applications which, in turn, can be designed according to any arbitrary parallel scheme. Besides, PFQ provides software bindings and APIs to several programming languages (namely C, C++, and Haskell) as well as a fully featured adaptation layer to legacy applications based on the pcap library. The system performance is thoroughly assessed and proves that PFQ reaches top class performance by hitting full capture, transmission and processing rates on 10/40+ Gbps links in simple speed-test bench-marking scenarios as well as in practical network use cases.

The most significant success of PFQ is that of traffic distribution; an aspect neglected if not entirely ignored by any other framework for network programming. With an extensive set of packet steering algorithms and through a pure functional language, PFQ allows for fine-grained control over the flows of packets and how they get distributed to endpoints, providing a broad range of different kinds of coherence at the flow-level.

The primary purpose of PFQ is to enable the execution of more instances of the same application (or the same worker thread), by partitioning and distributing network packets in such a way that each thread of execution does not require any synchronization and can work in complete isolation and autonomy.

Although of fundamental importance, the distribution of traffic is not supported by interfaces at the highest level of network programming. For example, this is the case of the pcap library, which has full support for accelerated sockets but does not provide an adequate interface for fanout and leaves this task to the application. In particular, the current implementation lacks workload splitting capabilities, thus preventing simple multi-core traffic processing schemes in legacy applications.

To overcome this limitation, this thesis presents an extension of the `libpcap` interface that integrates the packet fanout support. The new library enables parallel processing for both legacy applications and new multi-threaded ones, through the use of an extended set of API, as well as utilizing suitable environment variables and configuration files. The experimental validation has been extensively carried out in several scenarios by using standard and accelerated capture sockets.

After acknowledging the possible difficulties that one can face during the development of a high-performance network application, chapter 4 presents a functional programming language suitable for writing network applications in a simple, safe and performing way.

In particular, the chapter presents a functional language for both stateless and stateful processing pipelines on multi-core platforms. The language presented is grounded in the theoretical framework of monads borrowed from Category Theory and provides a formal description of a generic processing machine for network traffic manipulation. This programming model allows the parallelism of processing pipelines thanks to the immutability of packets, enforced by the functional paradigm, and to a suitable state-aware traffic splitting across multiple computation resources. A few examples are reported in the chapter to exemplify the practical language usage.

The language is implemented at two distinct levels of the network stack. At kernel space, the pfq-lang, which was already introduced in the previous chapter, is explained in detail. The language is presented as an integral part of PFQ and is intended to provide a tool for configuring the lower-level of the framework to perform packet filtering, distribution, load-balancer and network loggers.

However, given the exceptional complication introduced by kernel space programming, pfq-lang is designed to implement only state-less computations that run over the network-device drivers. For this reason, the same functional principles have been applied to the user space, where another version of the language is proposed and implemented as eDLS (embedded Domain Specific Language) on-top-of the Haskell language: Enif-lang.

This second implementation is intended to add stateful support to language. The Enif-lang runtime is designed to manage a heterogeneous set of applications that are deployed to the various cores of a multi-processor architecture, to maximize overall system performance. The actual implementation takes advantage of the underlying PFQ socket, to load-balance – and possibly copy – the traffic to a whole set of applications (or multiple instances of them), each intended to run in perfect isolation, hence leveraging the state-aware packet steering capabilities of the underlying framework.

At the time of writing, the aggregation of applications has dwindled to the programmer, that is in charge of configuring the runtime by choosing the degree of parallelism and the way they get aggregated together (which applications run on which cores), manually. We are analyzing some algorithms and heuristics that could automate the deployment of multiple Enif-lang applications. Such algorithms could exploit the Manhattan distance – computed over the number of bits that differ in the masks that identify the states associated with the flow – as we already proved it is a valid metric to use within the K-means clustering algorithm.

The chapter then presents a series of examples in Enif-lang and ends with a flow-tracker intended to count the number of packets of each TCP stream captured. A measure is also performed to evaluate the efficiency of the given implementation for a hash table – based on cuckoo hash scheme – in term of the number of packets per second processable on a single core, by varying the amount of streams present in the network.

Chapter 5 introduces some use-cases, where the elements of network programming are used to speed up the performance of existing network applications or to perform measurements of certain bandwidth algorithms without disrupting the system.

The first use-case presented is that of OpenFlow Soft Switch. The chapter shows in details the acceleration of OFSS, providing an implementation that can forward more than 4 Mpps on a single core and scale to run on multiple cores. When compared to state of the art, the prototype does not shine in terms of absolute performance numbers, but it provides a fast prototyping tool that is easy to modify and adapt for the exploration of new MAT abstractions.

We demonstrate such flexibility by porting a stateful MAT abstraction – OpenState – to the accelerated OFSS implementation. While the porting was effortless, our OpenState implementation could run a proof-of-concept stateful firewall function with higher throughput than a vanilla Linux's iptables implementation. Furthermore, in the process of accelerating OFSS, we shared our experience with the PFQ framework. In particular, we identified in the PFQ's programmable steering and dispatching functions a useful tool to simplify, and speed-up, network function implementation.

In the second use-case, the chapter presents FDPA, a design for a packet forwarding pipeline to enforce approximate fair bandwidth sharing. FDPA is based on primitives common in data plane abstractions such as P4 and OpenFlow. Differently, from other approaches based on per-packet scheduling, the implementation and time complexity of FDPA does not depend on the maximum number of active users. We performed experiments on a 10 Gbit/s real testbed, using PFQ as a measuring instrument to evaluate the performance without perturbing the system. Results show that performance is close to that of an ideal DRR scheduler with dedicated per-user queues, FDPA instead does not need per-user queues. We identified a trade-off between fairness and throughput, in which the throughput is penalized when configuring FDPA for more fairness. Preliminary analysis shows that packet reordering is the cause of such effect. We identified potential solutions to such problem that we leave for future work.

Finally, we present a system in which a set of distributed probes is deployed to detect network anomalies. The architecture takes advantage of the LogLog counter mixed with the Reversible Sketch data structure, which has been shown to be valuable tools for the anomaly detection and network programming in general. Since the max-merge operation in LogLog implicitly solves the problem of not counting duplicated data, the result of the aggregation of every single sketch is equivalent to that constructed with the mediator observing the total traffic from a single point. It worth noticing that the reduced memory footprint of such data structures and counters makes it an efficient tool for distributed systems, as the data that travels over the network for being aggregated is compressed, trading certainty for time/space. In conclusion, to prove the correctness of the system, we present an efficient and composable C++ library, as basic building block of the proposed architecture. Such a library, which implements probabilistic counters and data structures, is presented in details along with simple use-cases and the performance evaluation in term of probability and memory pressure.

Bibliography

[1] The Bro network security monitor.

[2] Linux Enhanced BPF (eBPF) Tracing Tools.

[3] OFSoftSwitch. `https://github.com/CPqD/ofsoftswitch13`.

[4] Open Networking Foundation. https://www.opennetworking.org/.

[5] OpenFlow Archive. http://archive.openflow.org/.

[6] Packet buffers. `https://people.ucsc.edu/~warner/buffer.html`.

[7] Probabilistic Data Structure (PDS) library. https://github.com/awgn/pds.

[8] The Frenetic Project.

[9] Tstat: TCP STatistic and Analysis Tool.

[10] VPP: https://wiki.fd.io/view/VPP.

[11] PFQ wiki, https://github.com/pfq/pfq/wiki, 2014.

[12] D Scott Alexander et al. The switchware active network architecture. *Network, IEEE*, 12(3):29–36, 1998.

[13] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 5–16, 2015.

[14] Paul Barford, Jeffery Kline, David Plonka, and Amos Ron. A signal analysis of network traffic anomalies. In *In Internet Measurement Workshop*, pages 71–82, 2002.

[15] G. Bianchi, E. Boschi, F. Gaudino, E. A. Koutsoloukas, G. V. Lioudakis, S. Rao, F. Ricciato, C. Schmoll, and F. Strohmeier. Privacy-preserving network monitoring: Challenges and solutions. In *17th ICT Mobile & Wireless Communications Summit 2008*, 2008.

[16] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM CCR*, 44(2):44–51, 4 2014.

[17] Giuseppe Bianchi, Marco Bonola, Salvatore Pontarelli, Davide Sanvito, Antonio Capone, and Carmelo Cascone. Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing. *CoRR*, abs/1605.01977, 2016.

[18] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[19] N. Bonelli, S. Giordano, and G. Procissi. Network traffic processing with PFQ. *IEEE Journal on Selected Areas in Communications*, 34(6):1819–1833, June 2016.

[20] Nicola Bonelli. Pfq homepage: http://www.pfq.io.

[21] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. On multi—gigabit packet capturing with multi—core commodity hardware. In *Proc. of PAM'2012*, pages 64–73. Springer-Verlag, 2012.

[22] Nicola Bonelli, S Giordano, and Gregorio Procissi. Enabling packet fan-out in the libpcap library for parallel traffic processing. In *Proceedings of the Network Traffic Measurement and Analysis Conference*, TMA'17, pages 1–9, June 2017.

[23] Nicola Bonelli, Stefano Giordano, Gregorio Procissi, and Luca Abeni. A purely functional approach to packet processing. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 219–230, New York, NY, USA, 2014. ACM.

[24] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[25] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *ACM SIGCOMM '13*, ACM SIGCOMM '13, pages 99–110. ACM, 2013.

[26] Lothar Braun et al. Comparing and improving current packet capturing solutions based on commodity hardware. In *IMC '10*, pages 206–217. ACM, 2010.

[27] Jake D. Brutlag. Aberrant behavior detection in time series for network monitoring. In *Proceedings of the 14th USENIX conference on System administration*, pages 139–146, Berkeley, CA, USA, 2000. USENIX Association.

[28] CAIDA. Analyzing UDP usage in Internet traffic. `https://www.caida.org/research/traffic-analysis/tcpudpratio/`, 2009.

[29] C. Callegari, A. Di Pietro, S. Giordano, T. Pepe, and G. Procissi. The loglog counting reversible sketch: A distributed architecture for detecting anomalies in backbone networks. In *Communications (ICC), 2012 IEEE International Conference on*, pages 1287–1291, June 2012.

[30] C. Callegari, S. Giordano, M. Pagano, and G. Procissi. Opencounter: Counting unknown flows in software defined networks. In *to appear in Proc. of SPECTS 2015*, July 2015.

[31] C. Cascone, L. Pollini, D. Sanvito, and A. Capone. Traffic management applications for stateful sdn data plane. In *EWSDN*, 2015.

[32] Carmelo Cascone, Davide Sanvito, Luca Pollini, Antonio Capone, and Brunilde Sansò. Fast failure detection and recovery in sdn with stateful data plane. *International Journal of Network Management*, 27(2), 2017.

[33] Cisco Systems. Snort homepage: https://www.snort.org/.

[34] Cisco Systems. Cisco Visual Networking Index: Forecast and Methodology, 2016-2021, June 2016.

[35] Graham Cormode and S. Muthukrishnan. Holistic udafs at streaming speeds. In *In SIGMOD*, 2004.

[36] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58 – 75, 2005.

[37] Willem de Bruijn, Herbert Bos, and Henri Bal. Application-tailored i/o with streamline. *ACM Trans. Comput. Syst.*, 29(2):6:1–6:33, May 2011.

[38] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM CCR 19.4*, 1989.

[39] Luca Deri. PF_RING ZC (Zero Copy).

[40] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *ACM SIGOPS*, pages 15–28, New York, NY, USA, 2009. ACM.

[41] DPDK. Distributor module.

[42] DPDK homepage: http://dpdk.org.

[43] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *In ESA*, pages 605–617, 2003.

[44] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Forwarding path architectures for multicore software routers. In *Proc. of PRESTO '10*, pages 3:1–3:6, New York, NY, USA, 2010. ACM.

[45] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *SIGCOMM Comput. Commun. Rev.*, 28(4):254–265, 1998.

[46] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Blooming trees: Space-efficient structures for data representation. In *2008 IEEE International Conference on Communications*, pages 5828–5832, May 2008.

[47] D. Ficara, A. Di Pietro, S. Giordano, G. Procissi, and F. Vitucci. Enhancing counting bloom filters through huffman-coded multilayer structures. *IEEE/ACM Transactions on Networking*, 18(6):1977–1987, Dec 2010.

[48] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frederic Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *nalysis of Algorithms (AOFA)*, pages 127–1460, 2007.

[49] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31:182–209, September 1985.

[50] Francesco Fusco and Luca Deri. High speed network traffic analysis with commodity multi-core systems. In *Proc. of IMC '10*, pages 218–224. ACM, 2010.

[51] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 29–38, Washington, DC, USA, 2015. IEEE Computer Society.

[52] Andrew D Gordon and Kevin Hammond. Monadic i/o in haskell 1.3. In *Proceedings of the haskell Workshop*, pages 50–69, 1995.

[53] The MAWI Working Group. Packet traces from WIDE backbone. http://mawi.wide.ad.jp/mawi/.

[54] Arpit Gupta, Muhammad Shahbaz, Laurent Vanbever, Hyojoon Kim, Russ Clark, Nick Feamster, Jennifer Rexford, and Scott Shenker. Sdx: A software defined internet exchange. *ACM SIGCOMM*, 2014.

[55] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[56] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. mswitch: A highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ACM SOSR '15, pages 1:1–1:13. ACM, 2015.

[57] Felipe Huici et al. Blockmon: a high-performance composable network traffic measurement system. *SIGCOMM Comput. Commun. Rev.*, 42(4):79–80, August 2012.

[58] Intel Corporation. Packet Processing. Intel DPDK vSwitch - OVS. `https://github.com/01org/dpdk-ovs`, 6 2015.

[59] Intel white paper. Improving Network Performance in Multi-Core Systems, 2007.

[60] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. Softflow: A middlebox architecture for open vswitch. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, USENIX ATC'16, pages 15–28. USENIX Association, 6 2016.

[61] Mike Jager. Securing ixp connectivity. In *In APNIC 34*, 2012.

[62] Raj Jain, Dah-Ming Chiu, and William R Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer system. *CoRR*, cs.NI/9809099, 1998.

[63] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM CCR 37.2*, 2007.

[64] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *ACM SOSR*, 2016.

[65] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18:263–297, August 2000.

[66] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247, New York, NY, USA, 2003. ACM Press.

[67] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *In ACM SIGCOMM*, pages 219–230, 2004.

[68] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proc. of ACM SIGPLAN-SIGACT*, pages 333–343, 1995.

[69] Linux Kernel. Huge Pages Documentation.

[70] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, USENIX NSDI'14, pages 459–473. USENIX Association, 2014.

[71] Paul E McKenney. Stochastic fairness queueing. In *IEEE INFOCOM*, 1990.

[72] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM CCR 38.2*, 2008.

[73] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[74] Nick McKeown et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[75] Eugenio Moggi. Computational lambda–calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society Press, 1988.

[76] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[77] V. Moreno, J. Ramos, P.M. Santiago del Rio, J.L. Garcia-Dorado, F.J. Gomez-Arribas, and J. Aracil. Commodity packet capture engines: Tutorial, cookbook and applicability. *Communications Surveys Tutorials, IEEE*, 17(3):1364–1390, thirdquarter 2015.

[78] Victor Moreno, Pedro M. Santiago Del Río, Javier Ramos, José Luis García Dorado, Ivan Gonzalez, Francisco J. Gomez Arribas, and Javier Aracil. Packet storage at multi-gigabit rates using off-the-shelf systems. In *Proceedings of the 2014 IEEE Intl. Conference on High Performance Computing and Communications*, HPCC '14, pages 486–489, Washington, DC, USA, 2014. IEEE Computer Society.

[79] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.

[80] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ACM HotSDN '14, pages 61–66. ACM, 2014.

[81] J. Nagle. On Packet Switches With Infinite Storage. IETF RFC 970, 1985.

[82] Ntop. PF_RING API.

[83] Ostinato Team. Ostinato: Packet traffic generator and analyzer.

[84] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. *ACM SIGCOMM CCR 33.2*, 2003.

[85] Rong Pan, Balaji Prabhakar, Flavio Bonomi, and Bob Olsen. Approximate fair bandwidth allocation: A method for simple and flexible traffic management. In *IEEE Allerton Conference on Communication, Control, and Computing*, 2008.

[86] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX OSDI'16. USENIX Association, 2016.

[87] Luca Petrucci, Marco Bonola, Salvatore Pontarelli, Giuseppe Bianchi, and Roberto Bifulco. Demo: Implementing iptables using a programmable stateful data plane abstraction. In *To appear in ACM SIGCOMM SOSR '17*, To appear in ACM SIGCOMM SOSR '17, 2017.

[88] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *USENIX NSDI '15*, USENIX NSDI '15, pages 117–130. USENIX Association, 5 2015.

[89] Phil Woods. libpcap mmap mode on linux.

[90] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[91] Lili Qiu, Yin Zhang, and Srinivasan Keshav. Understanding the performance of many TCP flows. *Computer Networks*, 37(3–4), 2001.

[92] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *INFOCOM, 2012 Proceedings IEEE*, pages 2471–2479, March 2012.

[93] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *Proc. of USENIX ATC'2012*, pages 1–12. USENIX Association, 2012.

[94] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ACM CoNEXT '12, pages 61–72. ACM, 2012.

[95] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the ACM SIGCOMM conference on Internet Measurement*, IMC '04, pages 207–212, New York, NY, USA, 2004. ACM.

[96] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *ACM SIGCOMM '16*, 2016.

[97] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX NSDI*, 2017.

[98] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 4(3), 1996.

[99] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM '16*, ACM SIGCOMM '16, pages 15–28. ACM, 2016.

[100] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *ACM SIGCOMM*, 2016.

[101] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SOSR*, 2017.

[102] SnabbCo. Snabb switch.

[103] SolarFlare. Openonload.

[104] Jonathan Philip Stringer, Qiang Fu, Christopher Lorier, Richard Nelson, and Christian Esteve Rothenberg. Cardigan: Deploying a distributed routing fabric. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 169–170, New York, NY, USA, 2013. ACM.

[105] Weibin Sun and Robert Ricci. Fast and flexible: Parallel packet processing with gpus and click. In *Proc. of ANCS '13*, pages 25–36, Piscataway, NJ, USA, 2013. IEEE Press.

[106] W. Szpankowski and V. Rego. Yet another application of a binomial recurrence order statistics. *Computing*, 43(4):401–410, 1990.

[107] The Wireshark Network Analyzer. tshark.

[108] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proceedings of the annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 615–624, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[109] M. Thottan and C. Ji. Anomaly detection in ip network. In *IEEE Trans. Signal Processing*, volume 51, pages 2191–2204, 2003.

[110] George Varghese. *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[111] Philip Wadler. The essence of functional programming. In *Proc. of ACM SIGPLAN-SIGACT*, pages 1–14, 1992.

[112] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. 1995.

[113] Shinae Woo, Lin Hong, and KyoungSoo Park. Scalable TCP session monitoring with symmetric receive-side scaling. Technical report, KAIST, 2012.

[114] Yin Zhang, Zihui Ge, Albert Greenberg, and Matthew Roughan. Network anomography. In *In IMC*, 2005.

[115] C V Zhou, Christopher Leckie, and Shanika Karunasekera. A survey of coordinated attacks and collaborative intrusion detection. *Computers Security*, 29(1):124–140, 2010.

[116] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ACM CoNEXT '13, pages 97–108. ACM, 2013.