Network Traffic Processing with PFQ

Nicola Bonelli, Stefano Giordano, *Senior Member, IEEE,* and Gregorio Procissi

***Abstract*—The paper presents PFQ, a high performance framework for packet processing designed to flexibly handle network applications parallelism and making traffic processing safe and easy. PFQ is an open–source module for the Linux kernel that combines software accelerated packet I/O to in–kernel early stage packet processing and fine–grained distribution to network applications and physical devices. PFQ does not require any modification to network device drivers and exposes programming interfaces to multi–threaded applications natively designed to run on top of it, as well as to legacy monitoring tools using the pcap library. The results show that the flexibility and backward compatibility provided by PFQ do not impact its processing performance that, in fact, reaches line rate figures in cases of pure speed tests and real practical monitoring use cases on 10+ Gbps links.**

***Index Terms*—Network Monitoring, Concurrent Programming, Multi–Core Architectures, Multi–Queue NICs, Early Stage Processing, Application Offload.**

I. Introduction

NETWORK applications in charge of performing any kind of processing on real data must be able to handle huge volumes [1] of heterogeneous traffic on high–speed (10+ Gbps) communication links. This is commonly the case of network monitoring applications, Intrusion Detection and Prevention Systems, routers, firewall and so on that must operate on–line with packet arrivals and process data in *streaming* mode to catch–up with traffic pace and promptly trigger the necessary operations.

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 is 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 actually 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.

This paper presents 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 [2] started a few years ago and was born as a Linux kernel capture engine [3]. Since then, the platform has changed quite a lot and many features have been added, including an in–kernel programmable stage for early processing [4]. 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 to legacy single process programs.

The reminder of the paper is organized as follows. Section II reports on the current background on traffic capture and monitoring and focuses on the specific motivations for using PFQ. Section III gives a high level view of the system and its logical design, while section IV delves into the details of the PFQ software acceleration. Section V describes the programmable layer of PFQ where early stage processing and distribution is performed, while section VI presents the physical and logical interfaces of PFQ towards the user–space world. Section VII describes PFQ transmission capabilities; the pure system speed is then assessed in Section VIII while a set of practical use–cases are provided in section IX along with their performance. Finally, section X concludes the paper.

II. 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

Nicola Bonelli, Stefano Giordano and Gregorio Procissi are with Dipartimento di Ingegneria dell'Informazione, Università di Pisa and CNIT, Via G. Caruso 16, 56122, Pisa, Italy e-mail: (nicola.bonelli@for., stefano.giordano@, gregorio.procissi@)unipi.it

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 general purpose operating systems, 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 [5] and more recently in [6] and [7].

PF_RING [8] 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] [9] and Netmap [10], 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 [11] that, besides accelerating traffic capture through OS bypassing, adds a set of libraries for fast packet processing on multicore architectures for Linux. OpenOnLoad [12] provides a high performance network stack to transparently accelerate existing applications. However, its use is strictly limited to SolarFlare products.

HPCAP [13] 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 the need for de–queueing packets at wire speed.

Packetshader [14] is a high performing software router that takes advantage of GPU power to accelerate computation/memory intensive functions. Egi *e*t al. [15] investigate on how to build high performance software routers by distributing workload across cores while Routebricks [16] 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 [17], a widely known framework that allows to build a router by connecting graphs of elements. Several works have recently focused on the acceleration of Clicks by means of some of the above listed I/O frameworks as in [18] and [19]. Furthermore, the Click approach has recently been complemented to take advantage of GPU computational power in Snap [20]. The Click modular principle is borrowed by Blockmon [21], a monitoring framework which introduces the concept of primitive composition on a message passing based architecture. Finally, the Snabb switch [22] 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 in order to support Receive Side Scale (RSS) [23] 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. In addition, the hash algorithm can be properly tweaked to achieve bi–directional flow coherency [24]. 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, in order to run multiple instances of the well known NIDS application Snort [25], a special symmetric hash function to achieve network–level coherency is required to properly detect cross–flow anomalies and attacks within a specific LAN.

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 [26], 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 (on the basis of a hashing scheme). In addition, 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 [27] 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 applications in order to implement of multi–threaded packet processing. However, such solutions are not fully transparent to the applications which, in turn, require to be adapted to take full advantage of these mechanisms.

As previously mentioned, the PFQ project started in 2011 and appeared first in [3]. Since its original version, PFQ was designed as a software accelerated capture engine with a basic in–kernel steering stage targeted at allowing user–space applications defining their arbitrary degree of parallelism.

[1]The successor of the formerly known PF_RING DNA

[2]The evolution of the formerly known libzero library

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 a number of 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 to hit 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.

In addition, 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, packets can be filtered, logged, forwarded, load-balanced and dispatched on a per–packet basis to application sockets, to generic endpoints or even to the kernel. This allows to run a specific accelerated PFQ application *concurrently* with a standard one (e.g., the standard OVS kernel module), even on the same NIC, *without changing a single line of code*. 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 paper 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 [4] and therein assessed *in isolation* only. In addition, 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.

III. The System Architecture

The architecture of PFQ as a whole is shown in Figure 1. In a nutshell, PFQ is a Linux kernel module designed to retrieve packets from *one or more* traffic *sources*, make some elaborations by means of 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

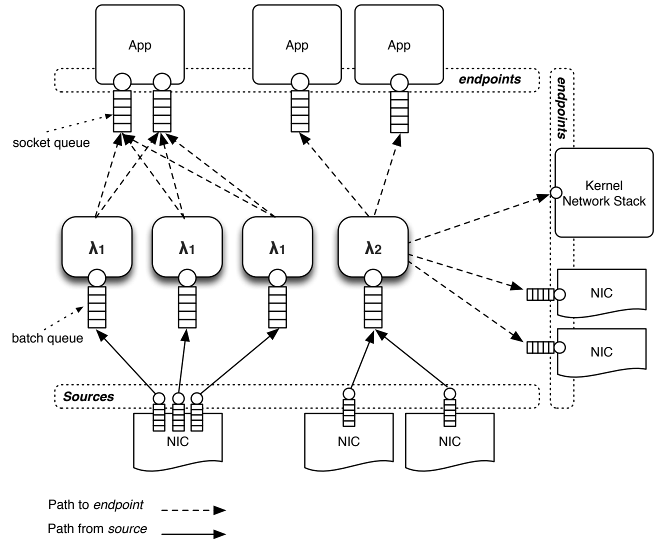


Figure 1. The PFQ system at–a–glance

multi–queue cards – by single hardware queues of network devices.

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 totally left under the OS control as the network device drivers are not modified.

A typical scenario for 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 simply stands along with it. Packets directed to networking applications are actually treated by PFQ exclusively, whereas packets destined to other system operations can be transparently passed to the kernel stack, on a per packet basis.

Figure 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 Haskell.

A. Three layers of parallelism

The system architecture depicted in Figure 1 clearly reveals three distinct levels of parallelism associated with three dif-

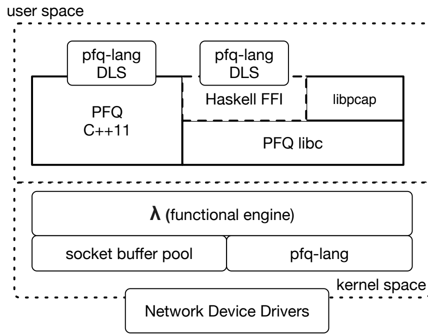


Figure 2. PFQ software stack

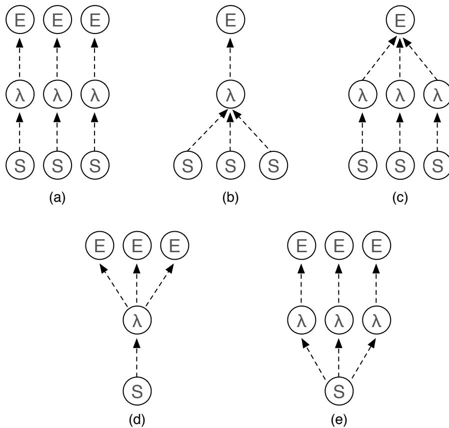


Figure 3. Three layers of parallelism

ferent 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 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 [23] to spread traffic among receiving queues by means of 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* in order to balance the overall capture load among the computation resources of the system.

In–kernel parallelism. Once interrupt affinity is set, capture parallelism is enabled and 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 paper 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 generally preserved as packet payloads might be transferred to different cores. This is the necessary price to pay to let user–space applications/threads run truly in parallel and perform stateful computations (e.g. per–user stats, per–mega flow processing, per–network monitoring, etc.) independently of each other.

Overall, hardware parallelism turns out to be totally 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 one hand, it allows a fine-grained control over the distribution process before packets are delivered to sockets without extra packet copies. On the other hand, it is completely transparent to applications (including legacy ones) which, in turn, do not require any modification to perform a proper parallel processing. In addition, the steering process brings up 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 it 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.

In addition, multiple logical schemes from Figure 3 may be instantiated at the same time as different applications may run concurrently in order 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 among each other making any application behave just as it were the only one running on the system.

IV. High Speed Packet Capture

This section describes the PFQ internals associated with the operations involved in low level packet capture. As above introduced, 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 normal driver data structures and operations. The impact of such acceleration techniques will be thoroughly assessed in the corresponding section of the performance evaluation results.

A. 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 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. In addition, the whole operation is made easier by the `pfq-omatic` tool included in the PFQ package that automates the compilation and only needs the original source code of the vendor device drivers.

B. 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 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 very short time, each pool contains enough recycled *skbuffs* to be reused upon driver request, so that the kernel allocator is no more needed.

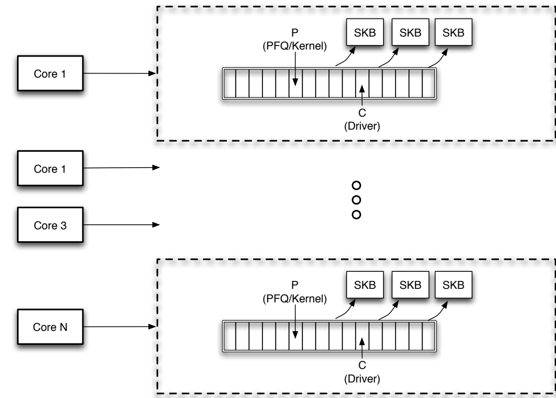


Figure 4. Pool of skbuffs

In queueing terminology, the pool can be modelled 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 are normally freed by the OS itself (through the standard `kfree()` function) after their use in the system.

C. 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. Essentially, 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 major 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 to decrease 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 within the performance section.

V. 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 endopoints 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 [4]. In addition, 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 totally 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 VI-B. 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 kernel or to NIC) and fanout functions (mainly, steering).

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

A. Groups and classes

One of the key feature 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 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). In addition, 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 to the same group receives 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 use 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 5 where a multi–threaded application is monitoring 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 process 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 broacast

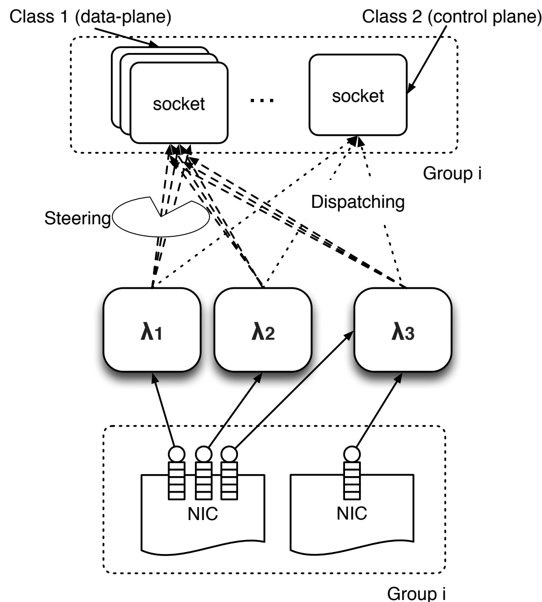


Figure 5. PFQ functional processing and fanout

(here called *Deliver*) mode or in load balancing (here called *Dispatch*) mode.

Coming back to the example of Figure 5, it comes out clearly that the combined use of groups/classes and the functional computation easily allow to fulfill 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 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 use of groups and classes will be provided in the use–cases section IX.

Groups access policy. Although the concept of 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.

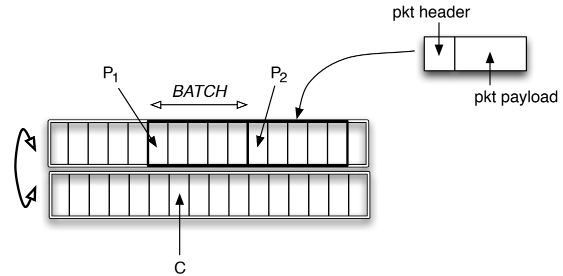


Figure 6. Double buffered socket queue

VI. User to Kernel space communication and APIs

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

A. 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 own queue; in the following they will be referred to as *socket queues*. In modern Linux systems, such queues are allocated on 2 MB large *hugepages* [28]; when this is not possible, PFQ automatically rolls back to standard system pages of 4 KB size.

Socket queues (Figure 6) are equipped with a double buffer. The use of a double buffer allows to decouple 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 start 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, buffer overflow may occur and packets are dropped.

Egress sockets. A specific discussion is needed when the final endpoint is not a socket, but rather a network device.

In such case, the socket queue is not used and the communications is implemented in a different way. To this aim, PFQ implements the special abstraction called *egress socket* that adapts the PFQ interface towards generic endpoints with no need to change the distribution operations.

B. 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 to program the kernel–space computations from both embeddable expressions (C++ and Haskell) and configuration files, by means of 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 qlang 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 easily 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, that convert functions that takes multiple arguments into functions that take a single argument. In addition, 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 [29] and Pyretic [30] in describing the data plane logic of an SDN network or to that of Streamline [31] 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) by means of 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 preserving class C network coherence.

Libpcap adaptation layer. Legacy applications using pcap library [32] 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, in order 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. In addition, 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 totally transparent to legacy pcap applications running on top of it. As such, for example, they can normally 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
```

VII. Packet transmission

Although the paper 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 with respect to the receiving side.

Socket queues are still doubly 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 by means of 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.

Throughout the rest of the paper, the transmission capability of PFQ is mainly used in the experimental sections to feed the PFQ receiving node 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.

In addition, the packet transmission capability of PFQ is used to effectively accelerate the well known *Ostinato* traffic generator [33]. A detailed report of this practical application is provided in section IX-D.

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 clearly evidences the benefits that this feature brings to PFQ performance.

VIII. 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 with respect to bare performance, in order to be effective it must be able to reach high–speed capturing and processing figures, possibly at the expenses of a slight extra cost in terms of the amount of system resources needed (i.e., number of cores).

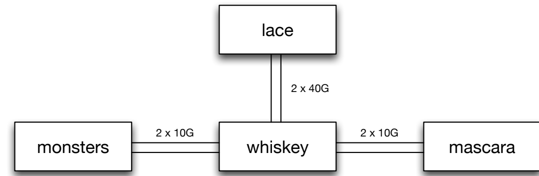


Figure 7. Experimental field trial

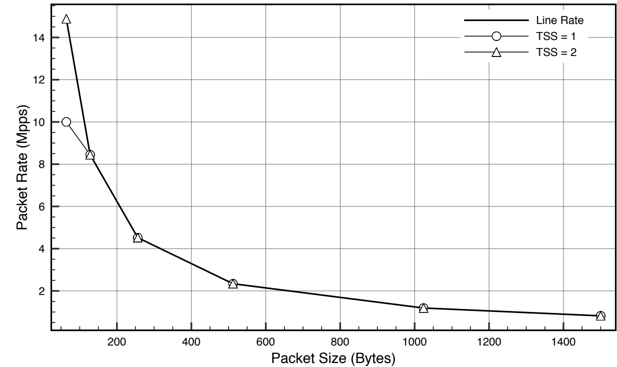


Figure 8. 10G packet transmission

The result reported in the following exactly demonstrates that PFQ allows commodity hardware to reach top class performance even by using pure vanilla drivers.

The experimental test bed used throughout the whole set of measurements is shown in Figure 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. In addition, two more Intel 82599 10G NICs were added to whiskey in order 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.

A. 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 VII, the user–space application in charge of generating packets and feed the PFQ transmission engines is `pfq-gen`, an open–source tool included in the PFQ distributon.

Figure 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).

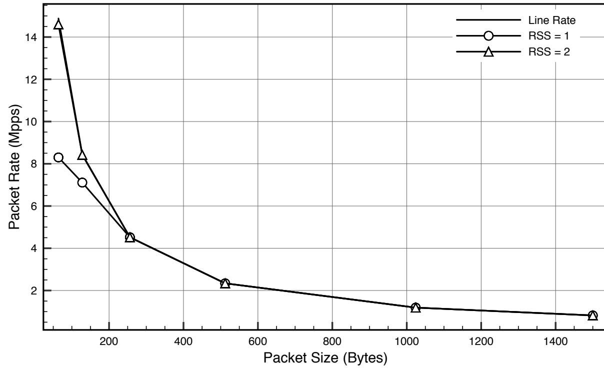


Figure 9. 10G packet capture: 1 user space thread

B. 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 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 number of hardware queues (i.e., number of cores used for capture).

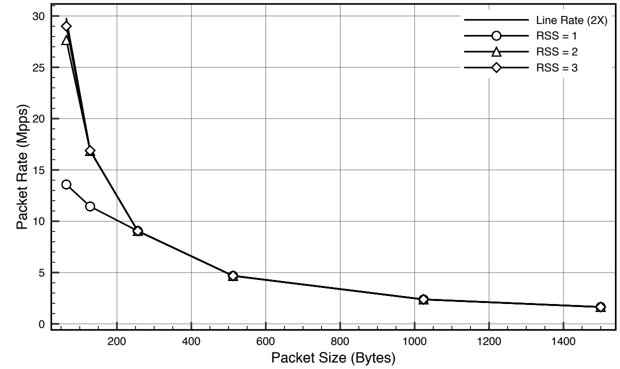
In the worst case of 64 bytes long packets, PFQ is capable of handling around 8.3 Mpps per core and, indeed, it requires two kernel engines (RSS=2) to reach line rate performance for all packet sizes.

When the application threads (belonging to the same monitoring group) become two (Figure 10), a slightly different number of kernel threads is needed due to the upstream delivery mode. Indeed, the broadcast mode (Figure 10(a))) requires PFQ to send an exact replica of all packets to both threads (which makes an internal throughput of 20 Gbps at full speed), while the steering mode (Figure 10(b))) 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 packets 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 for the extra core necessary with respect to the previous results of Figure 9.

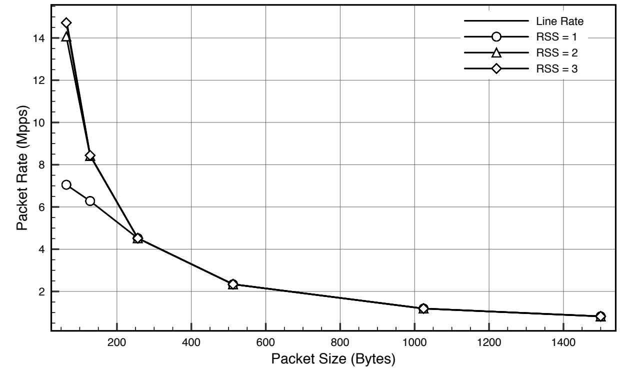
C. Up to 40G Speed Tests

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, a particular attention must be paid in the configuration of interrupt affinities (set



(a) Broadcast



(b) Steering

Figure 10. 10G packet capture: 2 user space threads

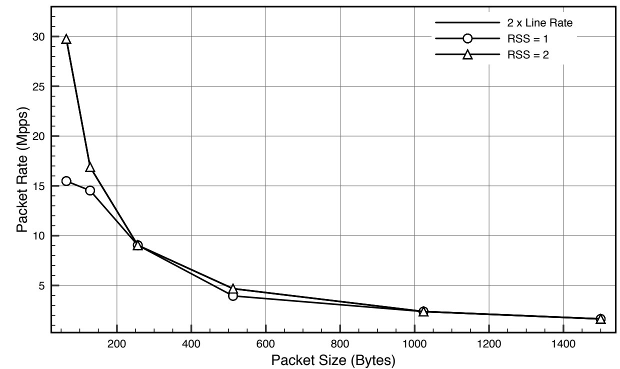


Figure 11. 20G packet capture

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 11 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 to the ones shown in Figure 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 2 cores.

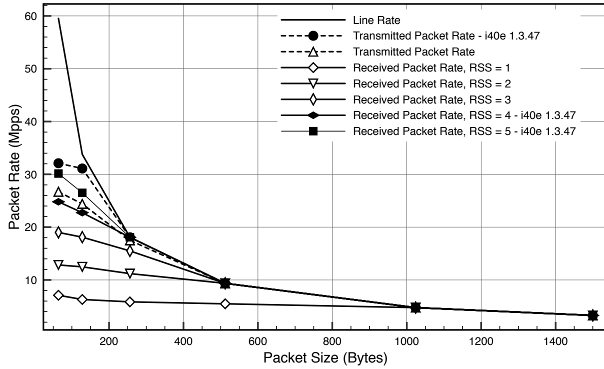


Figure 12. 40G packet capture

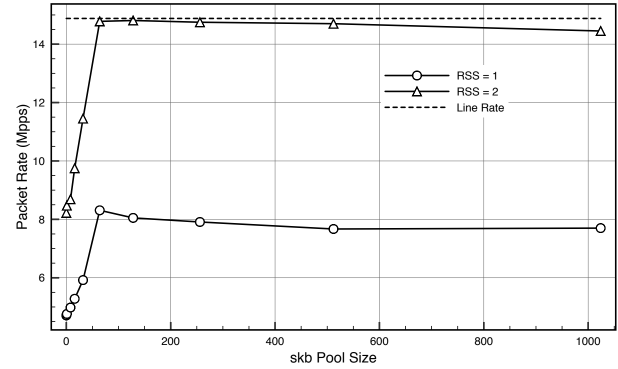


Figure 13. Skb pool acceleration

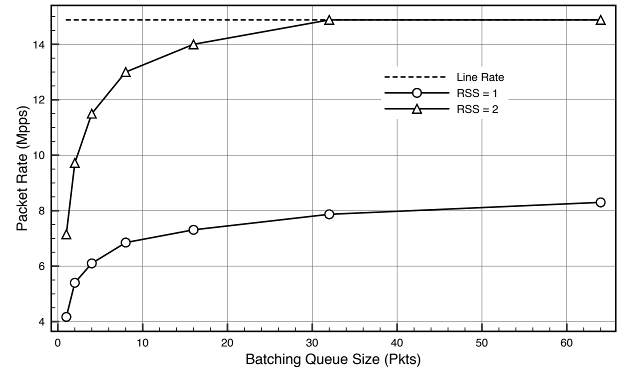


Figure 14. Batch queue acceleration

When link speed increases up to 40 Gbps, capture performance does not scale as well. Figure 12 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 Intel claim 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 than 256 bytes. By carefully looking at the figure, an other 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 totally 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 for 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 and 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 main "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.

D. 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 IV and in section VII. The following experiments aim at evaluating the impact of such acceleration techniques on the overall PFQ performance.

Figure 13 shows the effectiveness of using the pool of `skbuffs` in boosting capturing performance. Interestingly, the achieved throughput increases up to a pool size 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 number, additional pages must be used and this may lead to a little performance decrease.

Similarly, Figure 14 depicts the beneficial effect of packet batching in the cases of one or two cores devoted to capture. 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 15. Once 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.

E. 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

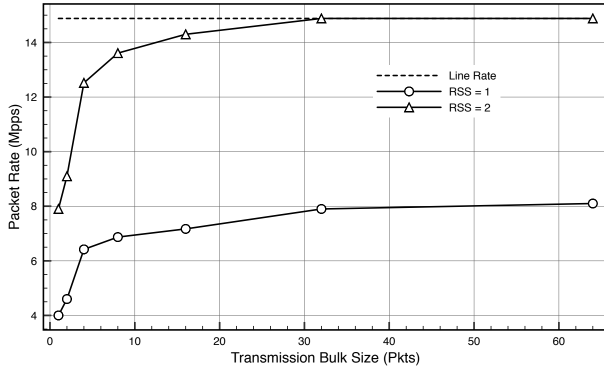


Figure 15. Bulk network packet transmission

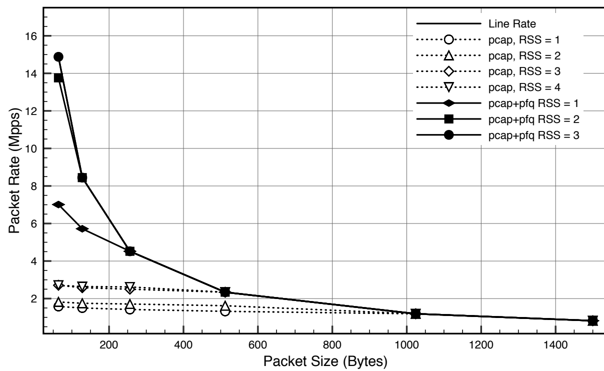


Figure 16. Libpcap acceleration

support provided by the RSS technology. Hence, the kernel–level capture system can be set to the parallel scheme of Figure 3(c) if the interrupt affinity is properly configured.

Figure 16 shows the results achieved by using a small pcap application that simply counts packets when running on top of the standard PF_PACKET socket and on top of PFQ, under different traffic packet sizes and different RSS values.

The performance improvement is evident and clearly shows that PFQ can effectively accelerate legacy network applications traditionally based on the pcap library. In addition, 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 4 hardware queues (RSS = 4).

The most likely reason for the observed libpcap performance resides in the implementation of the PF_PACKET socket, and specifically involves 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 properly configured), whereas PF_PACKET uses standard 4k system pages.

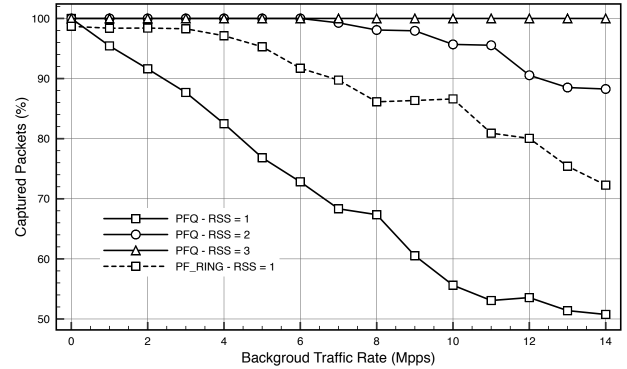


Figure 17. BPF filtered IP traffic with `tcpdump`

IX. 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* [33].

A. 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 by means of 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 17 shows (in solid lines) the number of packets received by the sniffer under different number of enabled hardware queues. Notice that RSS = 2 is sufficient to reach full rate filtering up to 6 Mpps disturb 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.

In addition, Figure 17 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 with respect to a well known alternative high–performing capture engine. The figure clearly 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

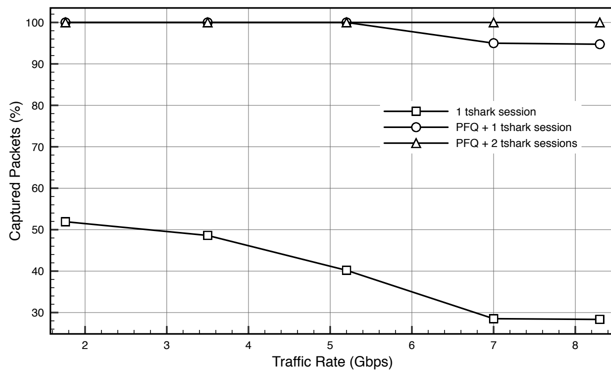


Figure 18. RTP flow processing with tshark

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.

B. *RTP flow analysis*

In the second use case, the pcap based tshark sniffer [34] (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 18 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 packet are not dropped by PFQ (which can easily handle 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. By means of PFQ, this problem can be elegantly overcome by increasing the number of user space instances of tshark and by let them join the same group. As a result, the top graph of figure 18 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 to spread RTP/RTCP traffic to multiple process instances, it still does not permit tshark to properly reconstruct flows and prepare a statistic summary. In fact, in order to accomplish such operations, both tshark instances need to access to SIP messages associated with the RTP flows. PFQ helps getting 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 18 are actually 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.

C. *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, with respect to the previous scenarios, this application represents a significant step ahead in benchmarking PFQ features and performance, in terms of 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 in the following, this definitely requires multiple threads to catch up with high traffic rates. In terms of 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. In addition, 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 by means of groups, classes and their configurable delivery mode for packet distribution.

The results shown in Figure 19 are obtained under different number of applications threads (sharing a common monitoring group and registered to a common control plane class) that receive:

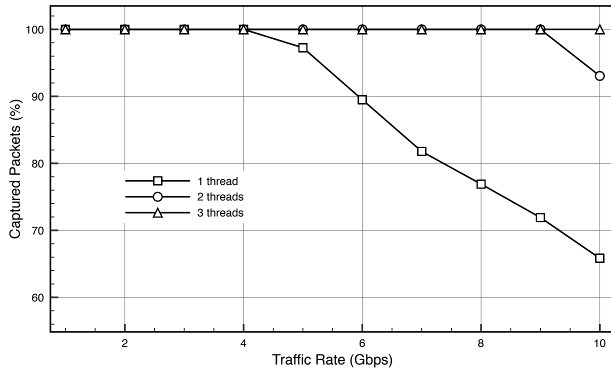


Figure 19. LTE analyzer

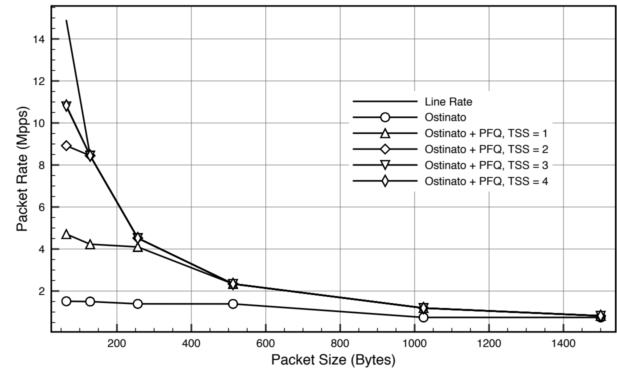


Figure 20. Ostinato packet transmission acceleration with PFQ

- 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 to those running the application threads. Figure 19 shows the percentage of received UP packets retrieved by the LTE analyzer application with respect to the input traffic rate and for 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 2 threads slightly suffer at full line rate only) to sustain a traffic rate of 10 Gbps.

D. 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 20 shows the result of the experiment. Ostinato was first run alone with the optimal value of 4 hardware queues for transmission (although, as shown in section VIII-E, 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 reaches near full rate generation speed in the only case of 1500 bytes long packets. In all other cases its performance is clearly 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 to bring the Ostinato performance above 10 Mpps transmission rate (i.e., yielding an acceleration factor slightly larger than 7) with 3 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.

X. Conclusion

The paper describes PFQ, an open source traffic 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 effectively handle traffic capturing and transmission 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. In addition, 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 pcap library.

The system performance has been thoroughly assessed and proves that PFQ reaches top class performance by hitting full capture, transmission and processing rates on 10+ Gbps links in pure speed–test benchmarking scenarios as well as in practical network use cases.

Acknowledgment

This work has been partly supported by the EU project BEBA and by the Italian MIUR project GreenNet.

References

[1] Cisco Systems, "Cisco Visual Networking Index: Forecast and Methodology," June 2011. [Online]. Available: http://www.cisco.com
[2] "Pfq." [Online]. Available: https://www.pfq.io/
[3] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi—gigabit packet capturing with multi—core commodity hardware," in *Proc. of PAM'2012*. Springer-Verlag, 2012, pp. 64–73.
[4] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni, "A purely functional approach to packet processing," in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '14. New York, NY, USA: ACM, 2014, pp. 219–230.

[5] L. Braun *et al.*, “Comparing and improving current packet capturing solutions based on commodity hardware,” in *IMC ’10*. ACM, 2010, pp. 206–217.
[6] V. Moreno, J. Ramos, P. Santiago del Rio, J. Garcia-Dorado, F. Gomez-Arribas, and J. Aracil, “Commodity packet capture engines: Tutorial, cookbook and applicability,” *Communications Surveys Tutorials, IEEE*, vol. 17, no. 3, pp. 1364–1390, thirdquarter 2015.
[7] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, “Comparison of frameworks for high-performance packet io,” in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 29–38.
[8] F. Fusco and L. Deri, “High speed network traffic analysis with commodity multi-core systems,” in *Proc. of IMC ’10*. ACM, 2010, pp. 218–224.
[9] L. Deri, “Pf_ring zc (zero copy).” [Online]. Available: http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/
[10] L. Rizzo, “Netmap: a novel framework for fast packet i/o,” in *Proc. of USENIX ATC’2012*. USENIX Association, 2012, pp. 1–12.
[11] “Dpdk.” [Online]. Available: http://dpdk.org
[12] SolarFlare, “Openonload.” [Online]. Available: http://www.openonload.org
[13] V. Moreno, P. M. S. D. Río, J. Ramos, J. L. G. Dorado, I. Gonzalez, F. J. G. Arribas, and J. 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*, ser. HPCC ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 486–489.
[14] S. Han, K. Jang, K. Park, and S. Moon, “Packetshader: a gpu-accelerated software router,” in *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, ser. SIGCOMM ’10. New York, NY, USA: ACM, 2010, pp. 195–206.
[15] 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*. New York, NY, USA: ACM, 2010, pp. 3:1–3:6.
[16] 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*. New York, NY, USA: ACM, 2009, pp. 15–28.
[17] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, “The click modular router,” *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, 1999.
[18] L. Rizzo, M. Carbone, and G. Catalli, “Transparent acceleration of software packet forwarding using netmap,” in *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 2471–2479.
[19] T. Barbette, C. Soldani, and L. Mathy, “Fast userspace packet processing,” in *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, 2015, pp. 5–16.
[20] W. Sun and R. Ricci, “Fast and flexible: Parallel packet processing with gpus and click,” in *Proc. of ANCS ’13*. Piscataway, NJ, USA: IEEE Press, 2013, pp. 25–36.
[21] F. Huici *et al.*, “Blockmon: a high-performance composable network traffic measurement system,” *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 79–80, Aug. 2012.
[22] SnabbCo, “Snabb switch.” [Online]. Available: https://github.com/SnabbCo/snabbswitch
[23] Intel white paper, “Improving Network Performance in Multi-Core Systems,” 2007. [Online]. Available: http://www.intel.it/content/dam/doc/white-paper/improving-network-performance-in-multi-core-systems-paper.pdf
[24] S. Woo, L. Hong, and K. Park, “Scalable tcp session monitoring with symmetric receive-side scaling,” KAIST, Tech. Rep., 2012.
[25] “Snort.” [Online]. Available: https://www.snort.org/
[26] Ntop, “Pf_ring api.” [Online]. Available: http://www.ntop.org/pfring_api/pfring__zc_8h.html
[27] DPDK, “Distributor module.” [Online]. Available: http://dpdk.org/doc/guides/prog_guide/packet_distrib_lib.html
[28] Linux Kernel, “Huge Pages Documentation.” [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
[29] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.
[30] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing software-defined networks,” in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482629
[31] W. de Bruijn, H. Bos, and H. Bal, “Application-tailored i/o with streamline,” *ACM Trans. Comput. Syst.*, vol. 29, no. 2, pp. 6:1–6:33, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1963559.1963562
[32] Phil Woods, “libpcap mmap mode on linux.” [Online]. Available: http://public.lanl.gov/cpw/
[33] “Ostinato: Packet traffic generator and analyzer.” [Online]. Available: http://ostinato.org/
[34] The Wireshark Network Analyzer, “tshark.” [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html

Nicola Bonelli received the master degree in Telecommunication Engineering from the University of Pisa, Italy. He is currently Ph.D. student at the Department of Information Engineering of University of Pisa. His main research interests are functional languages, software defined networking (SDN), wait–free and lock–free algorithms, transactional data–structures, parallel computing and concurrent programming (multi–threaded) on multi–core architectures. He collaborates with Consorzio Nazionale Inter-Universitario per le Telecomunicazioni (CNIT), and he is currently involved in the European Research project BEBA (Behavioral Based forwarding).

Stefano Giordano received the Masters degree in electronics engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 1990 and 1994, respectively. He is an Associate Professor with the Department of Information Engineering, University of Pisa, where he is responsible for the telecommunication networks laboratories. His research interests are telecommunication networks analysis and design, simulation of communication networks and multimedia communications. Dr. Giordano was chair of the Communication Systems Integration and Modeling (CSIM) Technical Committee. He is Associate Editor of the International Journal on Communication Systems and of the Journal of Communication Software and Systems technically cosponsored by the IEEE Communication Society. He is member of the Editorial Board of the IEEE Communication Surveys and Tutorials. He was one of the referees of the European Union, the National Science Foundation, and the Italian Ministry of Economic Development.

Gregorio Procissi received the graduate degree in telecommunication engineering and the Ph.D. degree in information engineering from the University of Pisa, Pisa, Italy, in 1997 and 2002, respectively. From 2000 to 2001, he was a Visiting Scholar with the Computer Science Department, University of California, Los Angeles. In September 2002, he became a Researcher with Consorzio Nazionale Inter-Universitario per le Telecomunicazioni (CNIT) in the Research Unit of Pisa. Since 2005, he is Assistant Professor with the Department of Information Engineering, University of Pisa. His research interests are measurements, modelling and performance evaluation of IP networks. He has worked in several research project funded by NSF, DARPA, European Union and Italian MIUR.