Packet Fan–Out Extension for the pcap Library

Nicola Bonelli, Fabio Del Vigna, Stefano Giordano, and Gregorio Procissi

***Abstract*—The large availability of multi–gigabit network cards for commodity PCs requires network applications to potentially cope with high volumes of traffic. However, computation intensive operations may not catch up with high traffic rates and need to be run in parallel over multiple processing cores. As of today, the vast majority of network applications – e.g. monitoring and IDS systems – are still based on the pcap library interface which, unfortunately, does not provide the native multi–core support, even though the current underlying capture technologies do.**

This paper introduces a novel version of the pcap library for the Linux operating system that enables transparent application level parallelism. The new library supports fan–out operations for both multi–threaded and multi–process applications, by means of extended API as well as by a declarative grammar for configuration files, suitable for legacy applications. In addition, the library can transparently run on top of the standard Linux socket as well as on other accelerated active engines. Performance evaluation has been carried out on a multi–core architecture in pure capture tests and in more realistic use cases involving monitoring applications such as *Tstat* and *Bro*, with standard Linux socket as well as PF_RING and PFQ accelerated engines.

***Index Terms*—Accelerated Sockets, Concurrent Programming, Multi-Core Architectures, Network Applications, Packet Fanout, pcap Library**

I. 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, such as Intrusion Detection and Prevention Systems, routers, firewall and so on. 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 [1] of heterogeneous traffic streams. In addition, the constant growth of computational power provided by the large number of cores available on affordable processors has favoured a significant 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 *packet I/O frameworks*, each of them with its own set of features. However, traffic capturing is only half of the task of traffic processing which, in fact, may require a significant (and, to some extent, orthogonal) phase of *packet analysis* thereafter. As a result, the higher packet rate attained by the accelerated capture engines may not, by itself, guarantee better application performance.

Indeed, computation intensive operations do not often catch up even with the low 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 paradigms and, on the other hand, the underlying capture technology to provide the support for *packet fan–out* 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 fan–out. Unfortunately, most of today's network applications are still single–threaded and access live traffic data through the Packet CAPture library (libpcap, or pcap in short) [2] rather than using the underlying raw sockets. Over the years, the pcap library has emerged as the de–facto standard interface for handling raw traffic data and, as it will be shown in the following, its use has many practical advantages. Examples of commercial and non—commercial applications using pcap include *Qosmos ixEngine* [3], *NetworkMiner* and *CapLoader* from Netresec [4], *Snort* [5], *Bro* [6], *Wireshark* [7], and so forth. However, the current pcap library does not support packet fan–out, thus preventing transparent application parallelism and hence requiring the applications themselves to implement the logic to load–balance the workload across multiple threads or processes.

This work presents the implementation of a new pcap library for the Linux operating system that supports packet fan–out while still retaining full backward compatibility with the current version. The new library is freely available for download[1] and provides an extended interface for network applications consuming live traffic data.

The paper extends the previous conference version [8] in several different directions. At first, the new pcap library itself has been extended with a set of APIs to simplify its use in practical scenarios. The applicability of the library has also been broadened to include the explicit support of a full set of accelerated sockets. In addition, the ongoing research towards

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

Fabio Del Vigna is with the Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Via G. Caruso 16, 56122, Pisa, Italy. E-mail: fabio.delvigna@for.unipi.it

[1]Repository at https://github.com/awgn/libpcap, branch "fanout".

a unified architecture for nearly–agnostic support of any type of underlying sockets is also given. The experimental part has been significantly extended by including new sockets in the performance tests as well as by adding more realistic traffic scenarios in the library assessment in practical use–cases.

More in detail, Section II presents the state–of–the art of software accelerated packet I/O, from the most popular low level capture engines to the higher level approaches for fast packet switching and routing. The standard scheme for accessing live network data on Linux is then provided in Section III, while Section IV presents a classification of the available accelerated capture engines into the two broad categories of *active* and *passive* sockets. The section specifically discusses the different issues that emerge when trying to enable packet fan–out 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 V briefly introduces the standard `pcap` interface by concisely reporting on its main features and the reasons of its popularity. Section VI represents the core of the paper and includes the description of the library for parallelizing native and legacy applications. Section VII 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 VIII and IX that report on the performance improvement brought by the new library in pure speed tests and in practical use cases involving the well known applications *Tstat* and *Bro*. Section X 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. Finally, Section XI concludes the paper.

II. Background

As previously discussed, the technological maturity reached by off–the–shelf hardware platforms has significantly fueled the research towards effective software accelerated solutions to packet capture. As a result, the echo–system of accelerated capture engines is nowadays quite populated.

At the low level, many approaches have been proposed to remove the limitations of general purpose operating systems. Many of them are designed to bypass the OS network stack if not the entire operating system. A detailed review of such approaches can be found in the papers [9], [10], [11] along with their comparison and usage guidelines.

One of the first software accelerated engines was PF_RING [12] 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) [13], and Netmap [14], 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 [15] 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. PFQ [16] is a software acceleration engine built upon standard network device drivers that primarily focuses on programmable packet fan–out. OpenOnLoad [17] rebuilds the network stack for SolarFlare products to seamlessly accelerate existing applications. HPCAP [18] is a packet capture engine that focuses on the efficient storage of live traffic into non–volatile devices and to perform timestamping and delivery to multiple listeners at user–space. NetSlices [19] is a framework developed at Cornell University which provides operating system abstractions towards hardware resources, such as multi–core processors and multi–queue network adapters. This allows fast packet processing at user–space with speed linearly increasing with the number of computational cores. The Linux kernel itself has significantly improved its capture performance with the adoption of the efficient TPACKET (version 3) socket that integrates PACKET_MMAP [20] that efficiently memory maps packets to user–space.

All of the above *frameworks* provide full control to low–level packet I/O, from the driver management (which can be vanilla or modified) to packet delivery to up–layer applications. The interaction between the underlying capture engine and the application developer is given by the specific set of APIs provided by the framework itself. To comply, most of such frameworks implement specific binding towards the `pcap` interface, the de–facto standard library for handling packet I/O. Indeed, the `pcap` library is used by the large majority of network applications, such as *tcpdump*, *wireshark*, *snort*, *Bro* and so on. However, when it comes to packet distribution to multiple workers, the `pcap` library lacks the explicit support for packet fan–out. Hence, packet distribution can only be enabled through either the low–level socket APIs or by taking advantage of the Received Side Scaling (RSS) hardware mechanism [21] as discussed in the works [22] and [23], or finally using additional libraries, such as the *Distributor library* of DPDK [24] and the commercial PF_RING ZC library[2] for PF_RING ZC.

The main contribution of this work is to extend the *pcap* library to support a unified packet fan–out mechanism over different capture–engines. This way, network applications will be able to select the packet distribution flavor straight from the `libpcap` API without the need for managing raw socket details and hardware configurations. As shown in Section IV, the extension applies to the general class of *active* sockets (which includes TPACKET, PF_RING, PFQ) and allows `pcap` based applications to split the workload across multiple threads/processes and to transparently replace the underlying capture engine within the active sockets.

For the sake of completeness, it is worth mentioning that beside the low level approaches, software acceleration has also been proposed at higher level in soft–based switches and routers, seldom taking advantage of the previously mentioned frameworks. Packetshader [25] was a successful proposal for a high performing software router that leverages GPU power to accelerate computation/memory intensive functions.

[2]Formerly known as `libzero` library.

It relies on a heavily modified driver that provides several optimizations, such as using a reduced version of the socket buffer structure and preallocating huge buffers to avoid per–packet memory allocations. Egi et al. [26] provide a thorough investigation on how to design and implement high performance software routers by distributing workload across cores.

A handful of proposals are based on the Click [27] modular router. Out of them, Routebricks [28] proposes an architecture to improve the performance of software–based routing by using multiple paths both within the same node and across different nodes by forming a routing cluster. The Netmap I/O framework has been used to accelerate Click [29] while FastClick [30] relies on the integration of both Netmap and DPDK and features I/O batching and advanced multi-processing techniques. Snap [31] improves the Click performance by offloading computation intensive processes to GPUs. PFQ has been used to accelerate the OpenFlow software switch OFSoftSwitch [32] and Blockmon [33], a monitoring framework that borrows the modular principle of Click and introduces the concept of primitive composition.

Finally, the Snabb switch [34] takes advantage of the kernel bypass mode of Ethernet I/O and of the Lua scripting language to build a fast and easy to use networking toolkit.

III. Packet Capture in Linux

In the Linux operating system, the whole mechanism of live traffic retrieval is initiated by the device driver that manages packets upon their arrival at the physical interface(s). Such packets are either made available to the network applications through the so–called *packet sockets*, or sent to the network stack of the operating system when targeting the machine itself. When used in *raw* mode, *packet sockets* provide native APIs for handling low level operations, such as opening/closing the socket itself, binding the socket to selected interfaces, selecting the dispatching method, and so on.

This section aims at describing the main internals of the default Linux capture socket with specific focus on the less known packet–dispatching features.

A. 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).

As shown in Figure 1, both TPACKET and AF_PACKET support multi–core packet capturing, that is they take advantage of the RSS algorithm to retrieve packets in parallel from multiple hardware queues of network interfaces.

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

Groups are implicitly created by the first socket joining a group and the maximum number of groups per network device is 65536. Yet, sockets join a fan–out group by means of the `setsockopt` system call with the `PACKET_FANOUT` option specified. Conversely, sockets can leave a group when closing. When the last socket registered to the group is closed, the group itself is deleted as well. Finally, to join an existing group, new packet sockets must obey the set of common settings already specified for the group, including the *fan–out mode*.

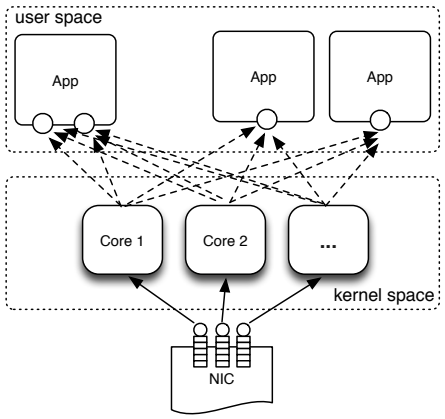

Fig. 1. Standard Linux socket

B. Socket Fan–out Modes

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

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

- The default mode, namely `PACKET_FANOUT_HASH`, preserves flow consistency by sending packets of 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 the endpoint to send the packet to.
- The `PACKET_FANOUT_LB` mode simply implements a round–robin 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.

IV. Software acceleration

So far, packet fan–out has been introduced as the straightforward way of scaling performance by splitting traffic workload among multiple workers, typically running on different cores. However, when link speeds raise up to multi–gigabit rates, the default Linux sockets may not be able to catch up with the actual packet arrival rate, causing a significant drop rate at the physical interfaces. A typical example is that of VoIP links of telco operators, in which significant packet rates, up to several millions packets per second, are common. In all such cases, the use of *accelerated capture sockets* is mandatory to increase the number of packets captured over the wire and dispatched to the application workers. Notice, however, that packet capture and distribution to up–layer endpoints are independent operations and even very efficient capture sockets may not necessarily support fan–out algorithms.

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 driver operations in the calling context – usually the user–space process. We name the sockets of the first category as *active sockets* since they require a running context (e.g., the NAPI kernel thread) to retrieve packets from the network device. Conversely, we name as *passive* the sockets that fall in the second category.

Among the accelerated sockets presented in Section II, the active sockets category includes the standard Linux PF_PACKET/TPACKET3, PFQ and PF_RING (both in its classic and aware flavors). Instead, PF_RING ZC, Netmap, and DPDK belong to the class of passive sockets.

Although under different names, all of the active sockets support packet fan–out in kernel space within the NAPI soft IRQ context. At this stage, the different implementations allow to distribute 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[3]. As a matter of fact, 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, including the implementation of receiving threads that poll the NIC and perform packet steering, the integration of lock–free queues for packets passing, etc. (incidentally, 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 fan–out feature with no modifications to the original source code of applications.

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

A. 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 of the PF_RING variants are supported by a custom `pcap` library (with no fan–out feature) implemented by the maintainers that makes it easily pluggable into legacy applications.

As shown in Figure 2, 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 in the user–space for application consumption. As such, PF_RING allows workload distribution to multiple rings (hence, multiple applications) and supports packet fan–out through the concept of *clustering*. PF_RING clusters are very similar to TPACKET and PFQ groups. Indeed, a set of applications sharing the same *cluster ID* receive packets coming from one or more ingress interfaces according to different balancing algorithms. Such algorithms typically rely on the aggregated values of selected IP header fields of either flat or tunnelled packets. As an example, the "round_robin" balancing scheme sends data to sockets according to round robin algorithm, while the "flow" algorithm delivers packets to sockets according to the hash value computed over the 6–tuple *<src ip, src port, dst ip, dst port, protocol, vlan tag>*.

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, as very top performance is left to the PF_RING ZC passive socket.

B. The PFQ accelerated socket

The architecture of PFQ as a whole is shown in Figure 3. In short, PFQ is a Linux kernel module that retrieves packets from *one or more* traffic *sources*, makes some *computations* by means of functional engines (the λ_i blocks in the picture) and finally delivers them 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.

Similarly to Linux sockets, PFQ uses the abstraction of group as the set of sockets that share the same computation and the same data sources. Each user–space thread or process opens a socket and *registers* the latter to a group. The group is then bound to a set of data sources and is associated

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

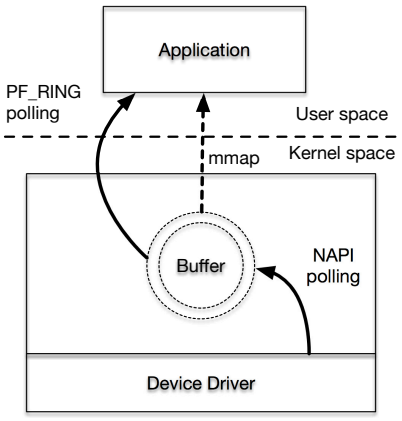

Fig. 2. The PF_RING socket

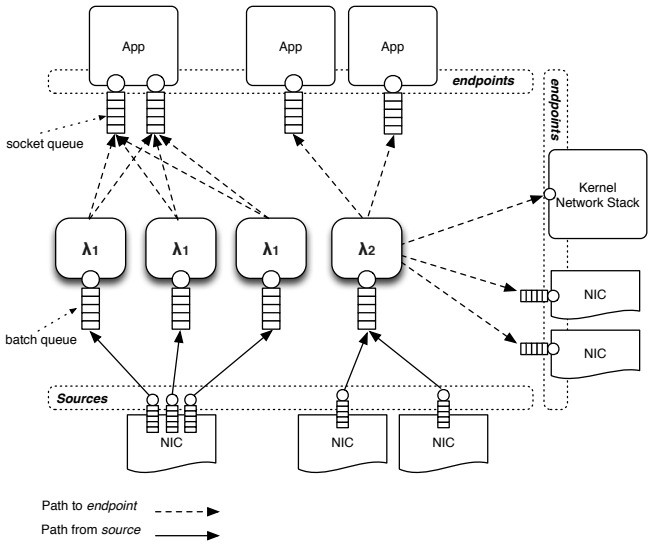

Fig. 3. PFQ–accelerated socket

with a functional computation instantiated as a PFQ–Lang program [35] that processes and steers packets among the subscribed endpoints.

V. The standard pcap interface

As above mentioned, packet sockets provide native APIs to let applications manage their low level operations. The very popular alternative to the use of native socket APIs is provided by the Packet CAPture (`pcap`) library. The `libpcap` layer stands logically between the socket level and the application level and provides standard and unified APIs for generic packet retrieval and handling. The `pcap` library is written in the C language and is available (as open–source) for Unix like operating systems as well as for Windows, hence easing application portability. The library provides a rich set of functions to handle live data and, with respect to native APIs, adds useful features such as read/write access to trace files and packet filtering by means of *Berkeley Packet Filters* (BPF).

The `libpcap` is widely adopted in commercial and non–commercial tools and, in fact, many popular network applications (such as *tcpdump*, *Wireshark*, *Tstat* etc.) and IDS (*Snort*, *Bro*) are written on top of it.

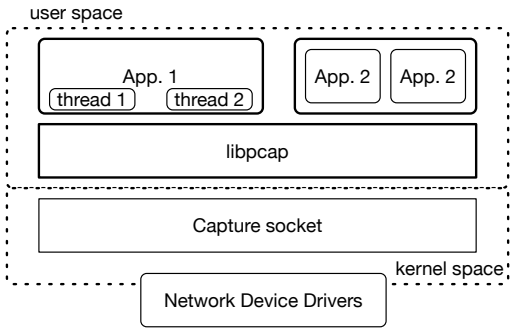

Fig. 4. Multi–workers network applications

However, as already mentioned, its major drawback is that it lacks a native support for multi–thread programming. This forces developers to implement their own parallel schemes (such as the one shown in Figure 4) to provide an additional layer of packet distribution right into the applications themselves. Indeed, in the example, both threads of the Application 1 would receive by default an exact replica of the same traffic, and so would the two instances of the Application 2. This design looks even more bizarre as the default socket used by `libpcap` in the Linux version (TPACKET) natively supports packet fan–out. The main objective of this work is indeed to remove this limitation.

VI. Packet fan–out support in the `pcap` interface

This section reports on the extension of the current `pcap` library that enables packet fan–out to provide flexible support for multi–core processing.

The starting point was to comply with the basic operation of the underlying Linux socket TPACKET so as to integrate the notions of *group of sockets* and *fan–out modes* into the `pcap` library. This implied a significant rework throughout the whole library code. However, as all of the changes are buried into the library implementation, the packet fan–out 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 takes the (integer) group identifier and a string representing the fan–out mode. It returns 0 in case of success and -1 when the operation cannot be completed[4].

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 fan–out mode.

When the extended library is used over the standard Linux socket, the fan–out mode should be selected among the ones listed in Section III-B. Conversely, when using an alternative socket, fan–out modes must comply with the ones supported by the underlying capture engine.

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

A. Legacy application: the pcap configuration file

The use of the extended API is well suited when writing a new application in a multi–threaded fashion. However, many popular network applications are single threaded and their refactoring toward a multi-threaded paradigm is not feasible in most practical cases.

In all such cases, the extended pcap library still allows to attain parallelism by running multiple instances of the same application and properly merging the outputs. Indeed, by using a declarative grammar specified in a *configuration file*, all of the processes that capture packets from the same NIC can join a common group and receive a fraction of the total traffic according to a specified fan–out algorithm, without any modifications to the source code.

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 fan–out mode (example: fanout = hash)
- caplen: integer 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
```

Different fan–out modes can also be selected for distinct 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 checks if the environment variable PCAP_CONFIG is set and, if so, it parses the file to retrieve the values of the keys therein specified.

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

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

will receive traffic according to the ”rnd” fan–out mode on the group 3, regardless of the values specified in the configuration file.

B. Accelerated configuration

The combined use of environment variables and the configuration file makes applications running totally agnostic to the underlying capture engine and to the way it implements the fan–out stage. As such, although the features of the capture sockets may be significantly different, the basic semantic of the pcap configuration does not change and the common set of environment variables reported in Table I can still be used irrespective of the underlying technology. However, socket–dependent features are still available by using the specific environment variables (e.g., those provided by the specific capture engines) supported for backward compatibility.

TABLE I
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 ”/etc/pcap.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 fan–out 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

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

PF_RING configuration.

The standard Linux socket can be effortlessly replaced with PF_RING by simply prefixing the names of the network devices to be monitored with the string “pfring” (as an example, pfring:eth3).

The semantic implemented by the new pcap library allows to select the fan–out algorithm and to choose the group number of the applications which is transparently mapped into the *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, within 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
```

It is worth noticing that even the PF_RING Zero Copy (ZC) passive socket is supported and it can be activated by simply prefixing the name of the network card with the string “zc”. However, as discussed in Section IV, packet fan–out is not available in this case as PF_RING ZC does not implement clustering at low level, unless a suitable RSS configuration is used.

PFQ configuration.

Similarly to PF_RING, the PFQ socket is enabled if the network device name is prefixed by the string "pfq". For applications that do not allow arbitrary names for physical devices (.e.g, when they expect real device names), its use can still be enabled by setting the environment variable `PCAP_DRIVER` to `pfq`, and this is true for any other kind of framework.

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 IV, the major benefit of using PFQ resides in its programmable fan–out described through the PFQ–Lang functional language. As such, the packet fan–out mode may indeed be specified through a PFQ–Lang program and conveniently placed in the configuration file as in the following example[5]:

```
# 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 let a process handle multiple devices at a time, each under a different group of sockets. A typical scenario is that of an OpenFlow Software Switch (e.g., *OFSoftSwitch* [36]), in which multiple instances of the switch can run in parallel by means of the new `pcap` library, each of them processing a separate 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
```

Here the application *tcpdump* sniffs traffic on the group 11 from device `eth0` and on the default group 42 from the device `eth1`.

Finally, there are cases in which an application needs to open the same device multiple times with different configuration parameters (e.g., with a different criterion for packet steering). In all such cases, the new library provides the concept of *virtual device*, namely a device name postfixed with the character ':' and with a number. This is very similar to the *alias* device name, but it does not require the user to create network aliases at system level. As an example, the next two lines allow to collect traffic from the network device `eth0` under two different group (11 and 23) by *virtually* renaming the network interface itself.

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

[5]Notice the use of the character > to prefix each line according to the *Haskell bird style* as alternative to the fan–out keyword.

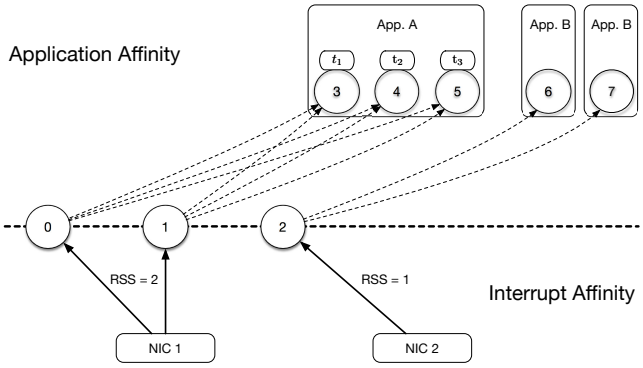

Fig. 5. Application and interrupt affinities

VII. `PCAP` FAN–OUT IN PRACTICE

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

By design, the code of the library is *re–entrant*, which means that it can be used in both single and 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 fan–out 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 to a dedicated group. This procedure allows it to capture the whole traffic coming from a NIC under the specified packet dispatching algorithm. Obviously, the same reasoning applies to 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 fan–out stage to deliver packets to a different number of end–points. While this is generally an endearing feature, it may however raise flow consistency issues. For this reason, applications with strict requirements on flow consistency should be restarted to guarantee correct results when the number of endpoints changes.

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 fan–out algorithms and degrees of parallelism.

A. Applications and Interrupt Affinities

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

Such a scenario is depicted in Figure 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 cores (possibly of the same NUMA node) to take advantage of maximum computation power. In any case, the application itself is ultimately responsible of setting its own 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.

The bottom part of Figure 5 shows another critical aspect to be addressed when dealing with packet capturing using active sockets. Indeed, in addition to user defined threads/processes, the Linux operating system provides a dedicated kernel thread (`ksoftirqd`) for packet capture. Such a thread is designed to run on top of any CPU serving the *soft interrupt (IRQ)* scheduled by network cards. Yet, the set of CPUs running the capturing kernel threads is defined by the *interrupt (IRQ) affinity*. Modern NICs (like 10/40G Intel cards) support multiple hardware queues (*channels*), where packets coming from the network are split into by using the RSS algorithm The distribution of traffic across such queues allows multiple `ksoftirqd` threads to fetch network packets in parallel, thus improving the receive performance of the system[6].

Both the number of channels[7] involved in the capturing operations, and 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 context on the same CPU is in general a bad idea. Conversely, wherever possible, the application affinity should be set to avoid core overlapping with the IRQ affinity. The latter can be set by means of rather naive bash scripts generally 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. To this purpose we provided a new set of APIs to let the application itself select on–the–fly the the number of channels and the IRQ affinity.

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);
```

The two APIs allow to set and get the number of hardware queues enabled for a given device. In particular, the `pcap_channels` data structure and its associated `channel_mask` allow the setter function to selectively update the number of the supported channels for device `dev`, namely *Rx, Tx, Combined* and *Other*. Conversely, the second function is used to retrieve the information about this number, as well as the type of channels enabled for the device. Nevertheless, depending on the hardware and the driver in use, some of the 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.

[6]An analogous behavior occurs in the transmission side.

[7]The number of channels is generally referred to as the RSS parameter, where $RSS = n$ means that n channels are used on that interface.

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 define the set of CPUs in charge of handling the IRQs and 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 following statement retrieves the full configuration for the device:

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

In analogous way, it is possible to specify the IRQ affinity to a specific set of CPUs for each 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 statements in the `pcap` configuration file, as in the example reported next:

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

VIII. 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 pairs of identical PCs with a 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.

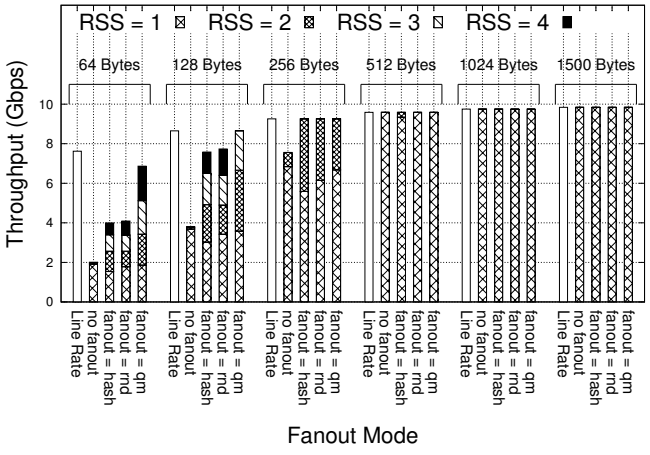

Fig. 6. 10 Gbps packet capture with `libpcap` over standard Linux socket

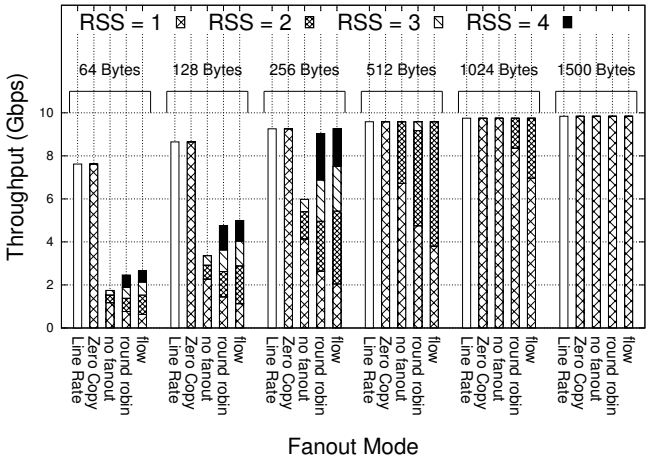

Fig. 7. 10 Gbps packet capture with `libpcap` over PF_RING

A. *Speed–Tests*

The first set of tests aims at assessing the impact of fan–out in the performance of the light–weight multi–threaded `pcap` application `captop`[8] 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 with randomized IP addresses are synthetically generated at 10 Gbps full line rate by `pfq-gen`, an open–source tool included in the PFQ distribution.

Figure 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 fan–out modes. The whole set of measurements is replicated by progressively increasing the number of underlying capturing cores, from 1 to 4 (RSS = 1,. . .,4), yet keeping 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 7 shows the results of the same test when the default Linux socket is replaced with PF_RING using the "flow" and the "round_robin" fan–out 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" driver once shipped 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 fan–out on the socket capture performance is observed.

For reference purposes, Figure 7 also 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 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

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

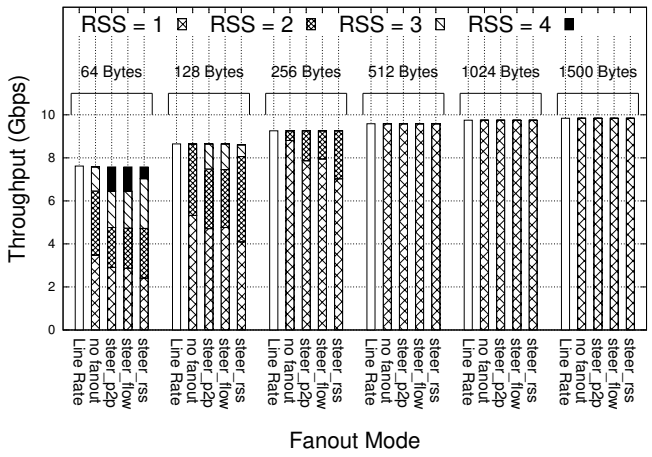

Fig. 8. 10 Gbps packet capture with `libpcap` over PFQ

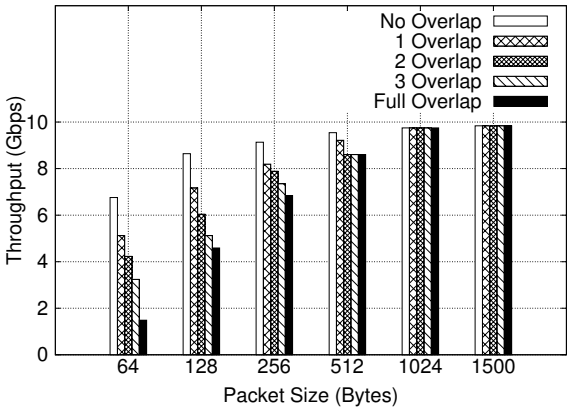

Fig. 9. Capture speed vs. application affinity

performance. This, in fact, is the expected effect of fan–out and it is consistently observed as the internal lock–free queues of PFQ efficiently manage multi–core access contention.

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 9 shows the capture speed 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.

IX. 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 it will be elaborated upon, in some cases the fan–out alone allows to scale the processing power up to full rate capacity while, in other 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. In the following, 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 means of socket acceleration. Conversely, the remaining metrics are associated with the application processing speed and can be improved by enabling packet fan–out.

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

A. Tstat

Tstat [37] is a popular tool for generic traffic analysis. It includes a large number of deterministic and statistical algorithms and can be used for 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 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 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 fan–out, two working instances of *Tstat* are sufficient to process all of the received packets.

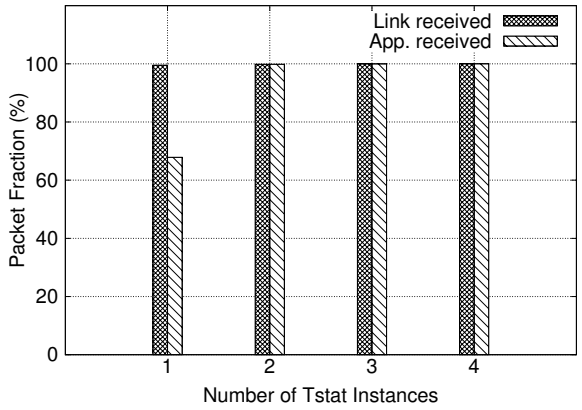

Fig. 10. Tstat and Linux socket: 10 Gbps traffic analysis with 300 Bytes average packet size

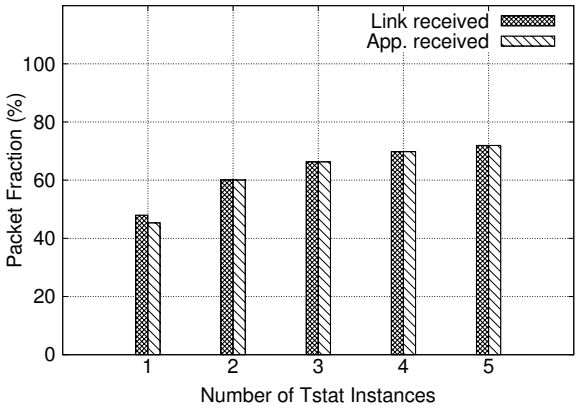

Fig. 12. Tstat and Linux socket: 10 Gbps traffic analysis with 128 Bytes average packet size

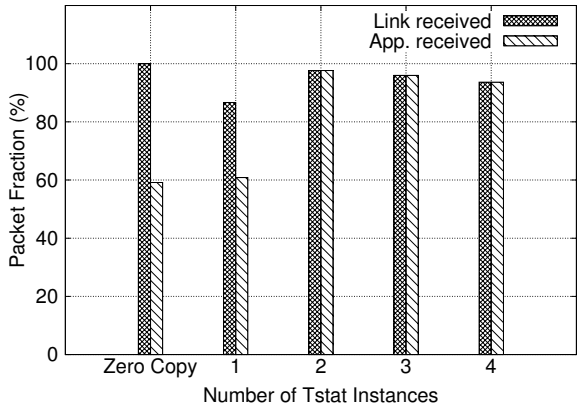

Fig. 11. Tstat and PF_RING: 10 Gbps traffic analysis with 300 Bytes average packet size

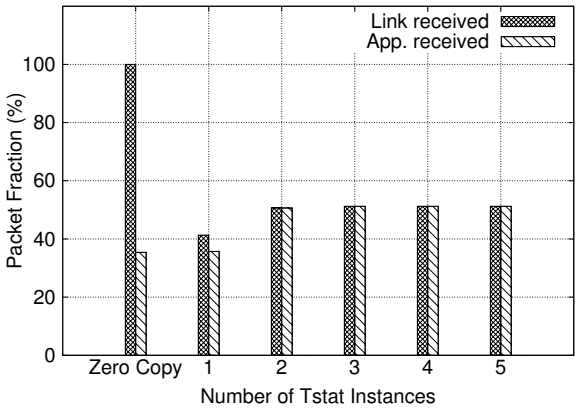

Fig. 13. Tstat and PF_RING: 10 Gbps traffic analysis with 128 Bytes average packet size

When TPACKET is replaced by PF_RING (Figure 11), the overall performance is somewhat similar, with two application instances capable of processing most of the offered traffic. 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, in spite of 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 12 and 13 report the results of TPACKET and PF_RING when running the same experiment with average packet size decreased to 128 Bytes (and corresponding average packet rate pushed up to 8.2 Mpps). In both cases, the use of fan–out allows two working instances of *Tstat* to effectively process all of the packets received on the physical device. However, nearly 40% and 50% of the input packets turns out to be dropped at the network interface as the input traffic rate exceeds the potential capture rates of the TPACKET and "classic" PF_RING, respectively. In addition, it can be noticed that the fraction of data processed by the application

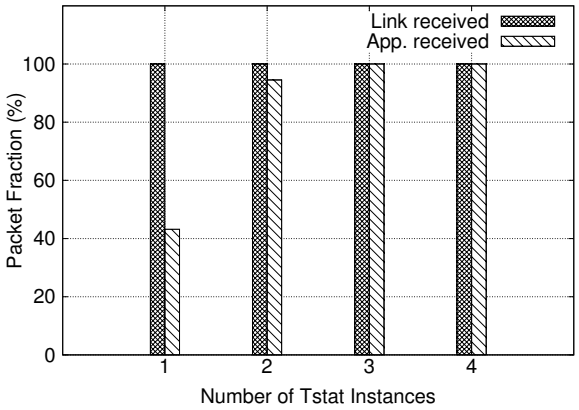

Fig. 14. Tstat and PFQ: 10 Gbps traffic analysis with 128 Bytes average packet size

is even lower in case of using PF_RING ZC, due to the CPU consumption required by the underlying packet capturing operations that run over the same CPU.

To further improve the performance of the application, packet fan–out can be conveniently combined with underlying socket acceleration. Indeed, as shown in Figure 14, the use of PFQ allows to avoid packet drop at the lower level and packet

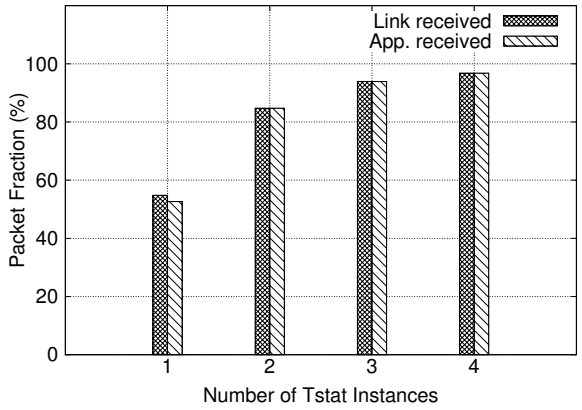

Fig. 15. Tstat and Linux socket: live traffic analysis of a real VoIP trace

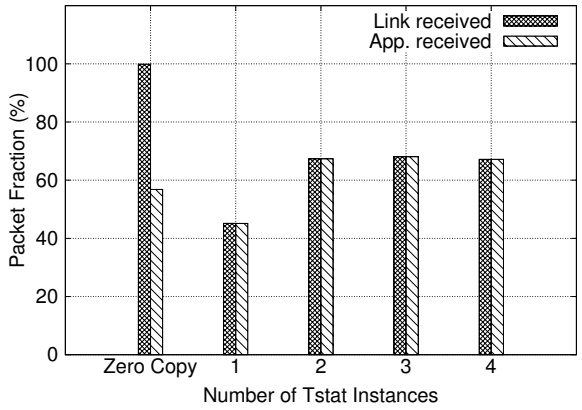

Fig. 16. Tstat and PF_RING: live traffic analysis of a real VoIP trace

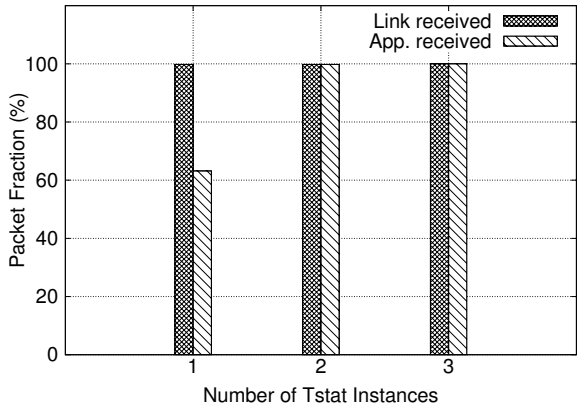

Fig. 17. Tstat and PFQ: live traffic analysis of a real VoIP trace

fan–out allows three instances of *Tstat* to successfully process nearly all of the input traffic.

In order to test the new library performance in a more realistic scenario, we run a further set of experiments in which *Tstat* is called to live process a real traffic trace that contains around 200K VoIP flows collected over a backbone network. Performance was recorded by feeding a varying number of *Tstat* instances with the VoIP packet data at around 5 Gbps speed. Figures 15, 16 and 17 show the results of the experiments when using TPACKET, PF_RING and PFQ capture sockets, respectively. In the first two cases, the underlying number of channels was set to four (RSS=4) in order to get the best possible performance, whereas using PFQ only two channels (RSS=2) were sufficient to fetch all of the packets from the network device. 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 further increased on our architecture without colliding to the underlying IRQ affinity. Conversely, PF_RING still exhibits performance saturation up to the second fetching cores, so as 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 so as two instances of *Tstat* are sufficient to process all the incoming traffic with two CPUs (RSS = 2) set to run packet fetching threads in the kernel space.

B. Bro

Analogous tests have been carried out to assess the performance of the *Bro* network security monitor [6] 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 plug–ins are required (currently, available plug–ins are available for PF_RING and Netmap sockets). The introduction of packet fan–out, instead, enables the use of the `libpcap` interfaces even in the cluster configuration by only setting a few environment variables with no need for extra plugins.

In the presented 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 hyper–threading technology was enabled when the number of *Bro* instances exceeded the number of physical cores.

Figure 18 shows the cluster performance when the standard Linux socket was used with two underlying capturing cores

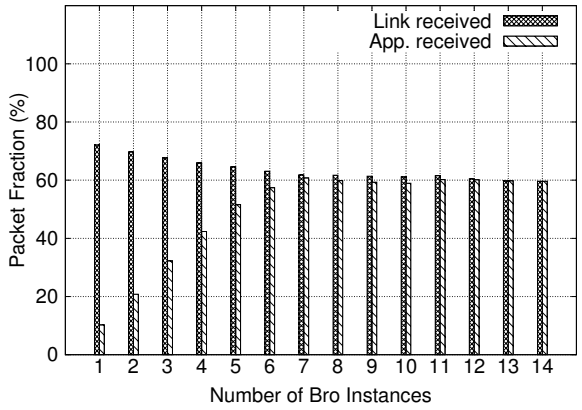

Fig. 18. Bro: real traffic analysis with standard Linux socket

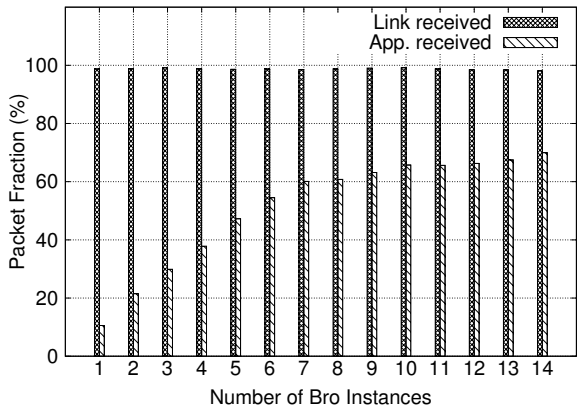

Fig. 19. Bro: real traffic analysis with PFQ

(RSS=2). The beneficial effect of fan–out is clearly 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 significant (up to 40%) and raises the need for socket acceleration.

Indeed, Figure 19 shows the results obtained when the standard socket is replaced by PFQ under the same number (RSS=2) of two 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 linearly, but the slope is reduced as the additional cores available through the hyper–threading 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 limits the maximum cluster cardinality to 14 nodes, as two of the overall 16 available cores were dedicated to underlying capturing/steering operations.

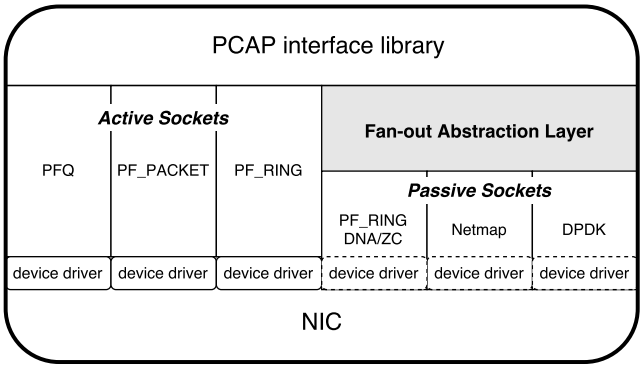

Fig. 20. The Fan–out Abstraction Layer

X. Towards the integration of passive sockets: the Fan–out Abstraction Layer

In Section IV we have discussed the complexity of providing the `pcap` library with the packet fan–out 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 passive sockets in the family of fan–out enabled socket supported by the new `pcap` library. Figure 20 depicts the complete scheme of the `pcap` library as we envision it. The `pcap` interface still stands in the top part as it is made of a set of functions that are invoked by 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 of sockets require a 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 data packets and apply the fan–out algorithms. We named this layer as *Fan–out 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 fan–out abstraction layer is still under development and its design is presented here as ongoing research.

In short, the role of the FAL is to hide the underlying machinery of passive sockets by exposing to the upper layer an *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 source code. As such, the FAL layer is responsible of translating the virtual directives of the FAL socket into real operations made onto actual sockets (Netmap, DPDK, etc.). Hence, the FAL must implement a set of *active pollers* that fetch packets from the network interfaces, apply the requested packet fan–out algorithm, and finally deliver packets to the sockets.

Under the above assumptions, the minimal set of APIs exposed by the FAL includes four basic classes of functions for *opening/closing* the FAL socket, *managing fan–out groups and algorithms*, *attaching/detaching* the `FAL` socket to physical network devices and implementing classical I/O primitives for *receiving and transmitting packets*.

Finally, as the system is intended to support both threads and processes, the implementation of the FAL is designed to store the configuration data in a shared memory that could conveniently be accommodated in Linux HugePages [38] for performance reasons.

XI. Conclusion

In spite of its widely common use in network applications, the current implementation of the `pcap` library lacks of workload splitting capabilities, thus preventing multi–core traffic processing schemes in legacy applications. This paper presents an extension of the `libpcap` interface for the Linux operating system that integrates packet fan–out support. The new library enables both native application multi–threading through the extended API as well as transparent multi–core acceleration for legacy applications by means of suitable environment variables and configuration files. The experimental validation has been extensively carried out in several scenarios by using standard and accelerated capture engines from the family of active sockets.

References

[1] Cisco Systems, "Cisco Visual Networking Index: Forecast and Methodology," June 2017. [Online]. Available: http://www.cisco.com

[2] Phil Woods, "libpcap mmap mode on linux." [Online]. Available: http://public.lanl.gov/cpw/

[3] "Qosmos." [Online]. Available: https://qosmos.com/

[4] NetResec, "Network forensics and network security monitoring." [Online]. Available: http://www.netresec.com/

[5] "Snort." [Online]. Available: https://www.snort.org/

[6] "The Bro network security monitor." [Online]. Available: https://www.bro.org/

[7] "The wireshark network analyzer." [Online]. Available: {https://www.wireshark.org}

[8] N. Bonelli, S. Giordano, and G. Procissi, "Enabling packet fan-out in the libpcap library for parallel traffic processing," in Proceedings of the Network Traffic Measurement and Analysis Conference, ser. TMA'17, June 2017, pp. 1–9.

[9] L. Braun et al., "Comparing and improving current packet capturing solutions based on commodity hardware," in IMC '10. ACM, 2010, pp. 206–217.

[10] 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.

[11] 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.

[12] 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.

[13] L. Deri, "PF_RING ZC (Zero Copy)." [Online]. Available: http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/

[14] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in Proc. of USENIX ATC'2012. USENIX Association, 2012, pp. 1–12.

[15] "DPDK." [Online]. Available: http://dpdk.org

[16] N. Bonelli, S. Giordano, and G. Procissi, "Network traffic processing with PFQ," IEEE Journal on Selected Areas in Communications, vol. 34, no. 6, pp. 1819–1833, June 2016.

[17] SolarFlare, "Openonload." [Online]. Available: http://www.openonload.org

[18] 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.

[19] T. Marian, K. S. Lee, and H. Weatherspoon, "Netslices: Scalable multi-core packet processing in user-space," in 2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Oct 2012, pp. 27–38.

[20] Linux Kernel Contributors, "Packet_mmap." [Online]. Available: https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt

[21] 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

[22] S. Woo, L. Hong, and K. Park, "Scalable TCP session monitoring with symmetric receive-side scaling," KAIST, Tech. Rep., 2012.

[23] M. Trevisan, A. Finamore, M. Mellia, M. M. Munafò, and D. Rossi, "Traffic analysis with off-the-shelf hardware: Challenges and lessons learned," IEEE Communications Magazine, vol. 55, pp. 163–169, 2017.

[24] DPDK, "Distributor module." [Online]. Available: http://dpdk.org/doc/guides/prog_guide/packet_distrib_lib.html

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ser. ANCS '15, 2015, pp. 5–16.

[31] 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.

[32] N. Bonelli, G. Procissi, D. Sanvito, and R. Bifulco, "The acceleration of ofsoftswitch," in 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Nov 2017, pp. 1–6.

[33] 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.

[34] SnabbCo, "Snabb switch." [Online]. Available: https://github.com/SnabbCo/snabbswitch

[35] 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.

[36] "OFSoftSwitch," https://github.com/CPqD/ofsoftswitch13.

[37] "Tstat: TCP STatistic and Analysis Tool." [Online]. Available: http://tstat.polito.it/

[38] Linux Kernel, "Huge Pages Documentation." [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

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).

Fabio Del Vigna received the master degree in Computer Engineering from the University of Pisa, Italy. He is currently a Ph.D. student at the Department of Information Engineering of the University of Pisa and collaborates with the Institute of Informatics and Telematics (IIT) of CNR, Pisa. He develops software for Social Media Mining and data analysis and worked on several national and European projects, including Cassandra, #toscana15, and CRAIM.

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 projects funded by NSF, DARPA, European Union and Italian MIUR.