Enabling Packet Fan–Out in the `libpcap` Library for Parallel Traffic Processing

Nicola Bonelli, Stefano Giordano and Gregorio Procissi
Dipartimento di Ingegneria dell'Informazione
Università di Pisa and CNIT
Via G, Caruso 16, 56122 Pisa, Italy
Email: {nicola.bonelli@for., stefano.giordano@, gregorio.procissi@}unipi.it

***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 are still based on the use of the `pcap` library interface which, unfortunately, does not provide a native multi–core support, even though the underlying capture technologies do.**

This paper introduces a novel version of the `pcap` library for the Linux operating–system that allows application level parallelism. The new library natively supports fanout operations for both multi–threaded and multi–process applications, by means of extended API as well as by a declarative grammar configuration suitable for legacy applications. In addition, the library can transparently run on top of the standard Linux socket and other accelerated capture 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 and the PFQ accelerated engine.

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

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

When the performance bottleneck is represented by the application itself, the straightforward way of scaling up performance is leveraging on *computational parallelism* by spreading out the total workload over multiple *workers* running on top of different cores. This, in turn, requires on one hand network applications to be designed according to multi–thread/multi–process paradigm and, on the other hand, the underlying capture technology to support *packet 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 `pcap` library (`libpcap`) [2] rather than using the underlying sockets. In the years, the `libpcap` library has emerged as the, somewhat, de–facto standard interface for handling traffic data and, as it will be shown in the following, its use has many practical advantages. However, the current `pcap` library does not support packet fan–out, thus preventing transparent applications parallelism.

The objective of this paper is to present 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 (e.g., intrusion detection systems, monitoring and security tools, traffic analysis platforms and so on). Therefore, the paper largely revolves around the concept of packet fan–out, in both its standard and accelerated declinations. At first, the standard scheme for accessing live network data on Linux is presented in section II. This includes a description of the

[1] https://github.com/awgn/libpcap-fanout

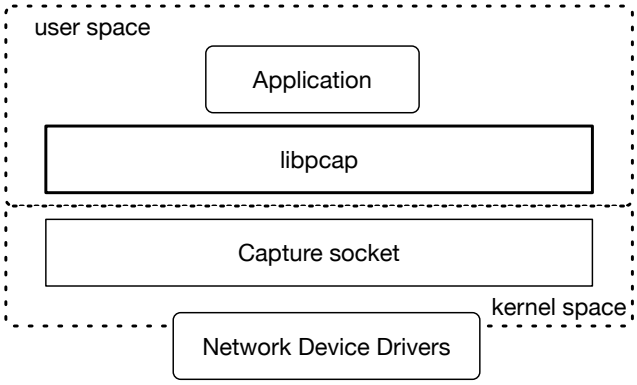

Fig. 1. Network application stack

standard Linux socket and its packet fan–out features as well as a brief introduction to the use of the `pcap` library.

Section III presents a short overview of the available accelerated sockets, with specific focus on PFQ as it will be used as an accelerated engine to further improve the capture performance of the newly developed library. Section IV represents the core of the paper and includes the description of the library for parallelizing native and legacy applications in both standard and PFQ–accelerated scenarios. Section V reports the results of pure speed tests carried out to numerically assess the benefit brought by the added fan–out support to the `pcap` library, while section VI elaborates upon the performance improvements observed by the well known applications `Tstat` and `Bro` when running in parallel on top of the new library. Finally, section VII concludes the paper.

II. Packet Dispatching in Linux

The typical scheme of a network application handling live traffic in the Linux operating system is shown in Figure 1. Upon their arrival at the physical interface, packets are managed by the device drivers and made available to the application through *packet sockets*. The low level handling of the socket operations can either be performed by the application through the native socket API or be left to the `pcap` library interface. This section aims at describing the main internals of the default Linux socket with specific focus on the less known packet dispatching features. The use of the `pcap` library is also briefly introduced to point out its current limitations that motivate this work in order to achieve a full integration with the standard socket 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).

At the lower level, both `TPACKET` and `AF_PACKET` support multi–core packet capturing, that is they take advantage of Received Side Scaling technology (RSS) [3] to retrieve packets in parallel from multiple hardware queues of network interfaces as shown in Figure 2.

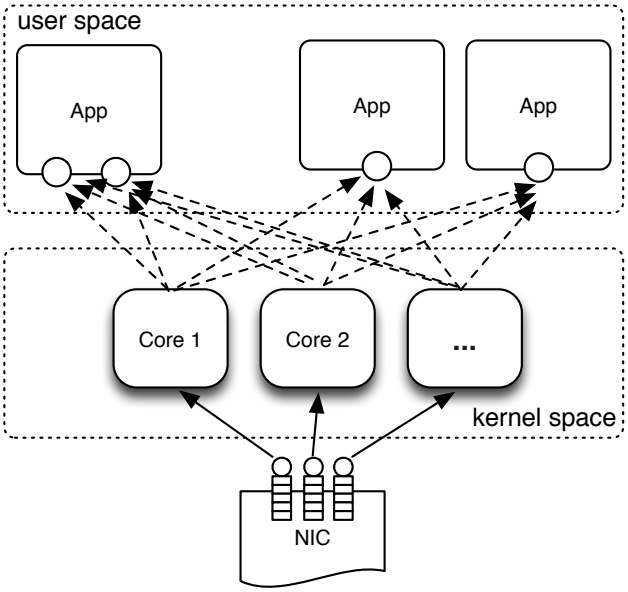

Fig. 2. Standard Linux socket

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

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

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 the workers have a significant impact in both functional and performance points of view.

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

- The default mode, namely `PACKET_FANOUT_HASH`, preserves flow consistency by sending packets from the same flow to the same packet socket. Practically, a hash

function is computed over the network layer address and (optionally) transport layer port fields. The result (modulo the number of sockets participating the group) is used to select the packet socket to send the packet to.
- The `PACKET_FANOUT_LB` mode simply implements a round–robin load–balancing scheme to choose the destination socket. This mode is suited for purely stateless processing as no flow consistency is preserved.
- The `PACKET_FANOUT_RND` mode selects the destination socket by using a pseudo–random number generator. Again, this mode only allows stateless processing.
- The `PACKET_FANOUT_CPU` mode selects the packet socket based on the CPU that received the packet.
- The `PACKET_FANOUT_ROLLOVER` mode keeps sending all data to a single socket until it becomes backlogged. Then, it moves forward to the next socket in the group until its exhaustion, and so on.
- The `PACKET_FANOUT_QM` mode selects the packet socket whose number matches the hardware queue where the packet has been received.

C. Standard pcap interface

Most of the more popular network monitoring applications (such as *tcpdump*, *wireshark*, etc.) are written on top of the `pcap` library [2]. As depicted in Figure 1, the `libpcap` layer hides low level traffic capture details to the upper layer application by providing a standard and unified API for generic packet retrieval and handling. As such, the use of `libpcap` eases application portability and adds useful features such as read/write access to trace files and packet filtering by means of *Berkeley Packet Filters* (BPF).

However, as a major drawback, the `pcap` library lacks the native support for multi–thread programming. This forces developers that need to implement schemes such as the one shown in Figure 3 to provide an additional layer of packet distribution built into the applications. By default, both threads of the Application 1 would receive an exact replica of the same traffic, and so would the two instances of the Application 2. This design looks even more paradoxical as the default socket used by `libpcap` in the Linux version (`TPACKET`) supports indeed packet fan–out. As will be elaborated upon in the following, the main objective of this work is to remove this limitation by providing the fan–out support to the `libpcap` interface.

III. Software acceleration

In the previous sections, packet fan–out has been introduced as the straightforward way of scaling performance by splitting traffic workload among multiple workers, tipically running on different cores or CPUs. However, when links speed raises to multi–gigabit rates, the default sockets may not be able to catch up with the actual packet arrival rate, causing a significant drop rate at the physical interfaces. In all such cases, the use of *accelerated capture sockets* is mandatory to increase the number of packets captured on the wire and dispatched to the application workers. Notice, however, that

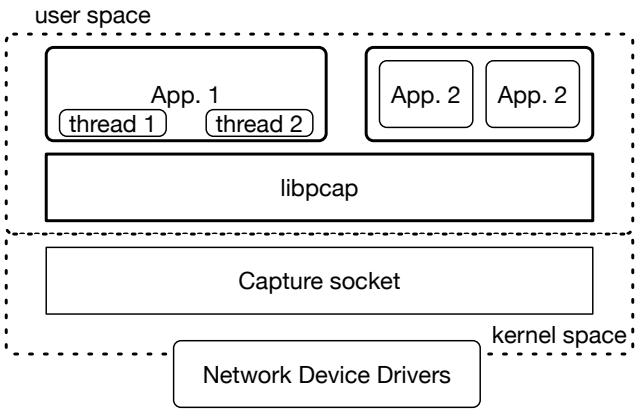

Fig. 3. Multi–workers network applications

packet capture and packet distribution to up–layer software are independedent operations and very efficient capture sockets may not necessarily support fan–out algorithms.

In the last few years, a significant number of accelerated sockets have been proposed for efficient traffic capture at 10G+ links speed (see references [4], [5], [6] for a thorough overview). One of the first software accelerated engines was PF_RING [7] 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 and modified drivers. More recently, PF_RING ZC (Zero Copy) [8], and Netmap [9], 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 [10] is another successful solution that bypasses the operating system to accelerate packet capture. DPDK provides a Linux user–space framework for efficient packet processing on multi–core architectures based on pipeline schemes. Finally, PFQ [11] is a software acceleration engine built upon standard network device drivers that primarily focuses on programmable packet fan–out. For these reasons, PFQ will be used in this paper to show how to run the new `libpcap` library on top of an accelerated socket that provides a richer set of fan–out options.

The architecture of PFQ as a whole is shown in Figure 4. In short, PFQ is a Linux kernel module that retrieves packets from *one or more* traffic *sources*, make some *computations* by means of functional engines (the λ_i blocks in the picture) and finally deliver 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 groups as the set of sockets that share the same computation and the same set of data sources. User–space threads/processes open a socket and *register* the socket to a group. The group is then bound to a set of data sources and is associated with a functional computation instantiated by a PFQ–Lang

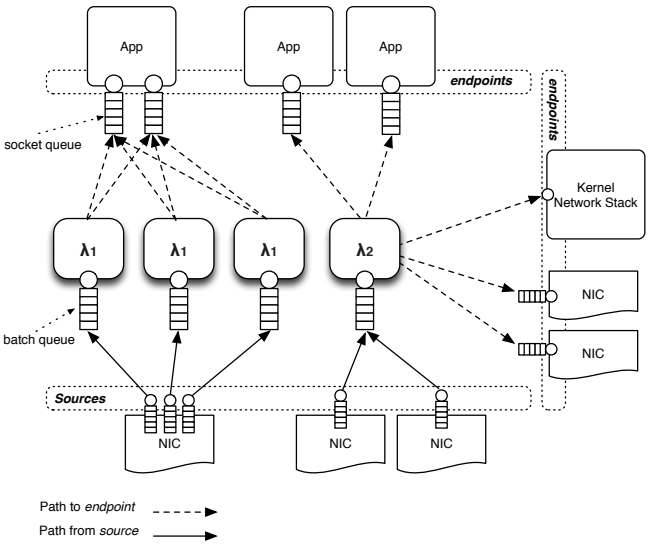

Fig. 4. PFQ–accelerated socket

program [12] that processes and steers packets among the subscribed endpoints.

Table I lists the main fan–out modes implemented in PFQ and reports (if any) the analogous fan–out mode of the standard Linux socket.

IV. Packet fanout support in the pcap interface

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

The starting point was to comply with the basic operation of the underlying Linux socket `TPACKET` by integrating the notions of *group of sockets* and *fan–out modes* into the `pcap` library. This implied a significant reworking throughout the whole library code. However, all of the changes are buried into the library implementation, and 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 requires to specify the (integer) group identifier and a string representing the fan–out mode. The function returns 0 in case of success and -1 in case the operation cannot be completed[2].

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

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 II-B and provided by the socket itself. When using an alternative socket, fan–out modes must comply with the ones supported by the underlying engine (see Table I for the main fan–out modes available with PFQ and the equivalent supported by `AF_PACKET/TPACKET`).

A. Legacy application: pcap configuration file

The use of the extended API is well suited when writing a new application in a multi–threaded fashion. However, most widely popular network applications are single threaded and their rewriting according to a multi–threading 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. All processes that join the same group will then receive a fraction of the total traffic workload, according to a declarative grammar specified in a *configuration file* and without requiring modifications to the application itself.

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

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

where the most commonly used keys are:

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

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

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

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

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

Notice that several keys of the configuration file can also be specified in the command line by means of additional environment variables, with the consequence of overriding the

TABLE I
PACKET FAN–OUT MODES IN PFQ AND THEIR LINUX SOCKET COUNTERPARTS

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

correspondent settings in the configuration file. As an example, a generic instance of the application `foo` launched as:

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

will receive traffic according to the ”rnd” 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 on top of the new `pcap` library totally agnostic to the underlying capture engine and to the way it implements the packet fan–out.

As an example, this section reports on the specific configuration needed to use of the `pcap` library on top of the PFQ socket. It is worth pointing out that analogous arguments may be applied to other accelerated capture engines such as certain versions of PF_RING equipped with libzero, if properly integrated.

By default, PFQ socket is enabled whenever the network device name is prefixed by the string ”pfq”. However, for applications that do not allow arbitrary names for physical devices, it can still be enabled by specifying an environment variable with the name of the device prefixed by `PFQ_` (e.g. `PFQ_eth0=1`). This permits selectively choosing the devices that will be using PFQ sockets and the devices that will not. The environment variable `PFQ_FORCE_ALL` set to one, instead, forces the use of PFQ sockets for all devices.

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.

Since PFQ allows a higher configuration granularity, the number of environment variables is larger than that available for the standard `TPACKET` socket. For the sake of completeness, the full list of them is reported in Table II along with a short description for practical usage.

TABLE II
PFQ ENVIRONMENT VARIABLES

Environment Variable	Description
PFQ_CONFIG	Specify the PFQ/pcap config file
PFQ_FORCE_ALL	Force PFQ sockets for all devices
PFQ_DEF_GROUP	Specify the PFQ group for the process
PFQ_GROUP	Specify the PFQ group for the process
PFQ_CAPLEN	Override the snaplen value for capture
PFQ_RX_SLOTS	Define the RX queue length of the socket
PFQ_TX_SLOTS	Define the TX queue length of the socket
PFQ_TX_SYNC	Hint used to flush the transmission queue
PFQ_TX_HW_QUEUE	Set the TX HW queue passed to the driver
PFQ_TX_IDX_THREAD	Set the index of the PFQ TX kernel threads (optional)
PFQ_LANG_SRC	Load the PFQ–Lang computation from source file
PFQ_LANG_LIT	Set the PFQ–Lang computation from the env. variable
PFQ_VLAN	Set the PFQ vlan id filter list for the group

As previously introduced in section III, 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 by a PFQ–Lang program and placed in the configuration file as in the following example[3]:

```
# Pcap configuration file (PFQ flavor)

def_group  = 11
caplen = 64
rx_slots = 131072
```

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

```
>  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* [13]), in which multiple instances of the switch can run in parallel by means of the new `pcap` library, each of them processing a portion of the traffic over a set of network devices.

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

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

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

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

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

V. 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 and on top of 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.

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`[4] that simply counts the received packets when running on top of both the standard Linux socket and PFQ, under different packet sizes and number of underlying capturing cores (different RSS values). Packets are synthetically generated at 10 Gbps full line rate by `pfq-gen`, an open–source tool included in the PFQ distributon.

[4]Avaialble at https://github.com/awgn/captop

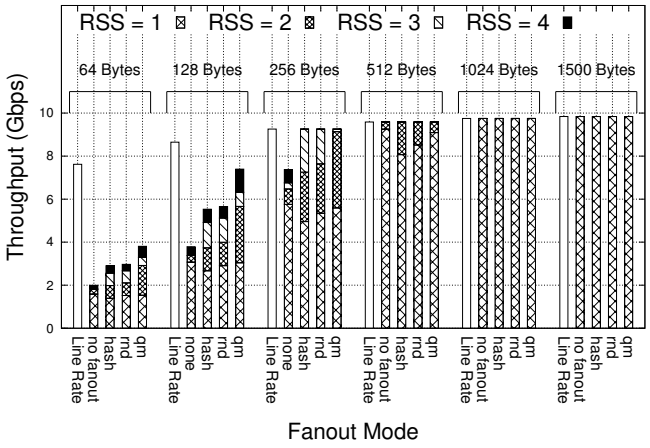

Fig. 5. 10 Gbps packet capture with `libpcap` over standard Linux socket

Figure 5 shows the result of the speed–test when *two working threads* of `captop` retrieve the packet streams on top of the `TPACKET` Linux socket according to three 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). 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 256 Bytes long packets. 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 concurrently inject packets to a single socket. With fan–out enabled, when the number of application sockets increases, the contention of the sockets queues among multiple producer contexts (*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. This is due to the different computational burden of each individual fan–out algorithm. 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, instead, need to either generate random numbers or compute hashes functions before dispatching packets to the target sockets.

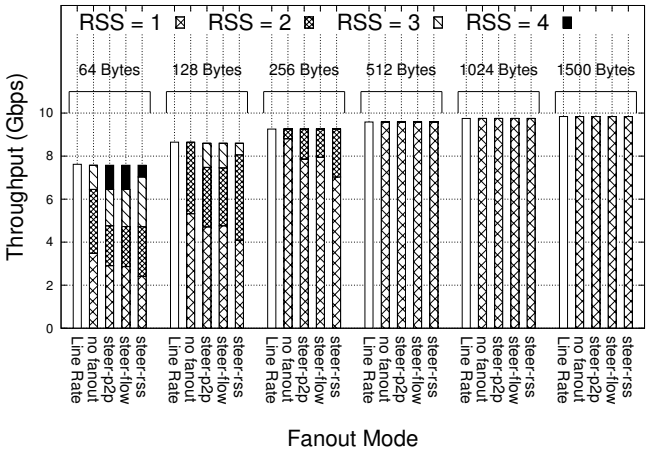

Fig. 6. 10 Gbps packet capture with `libpcap` over PFQ accelerated socket

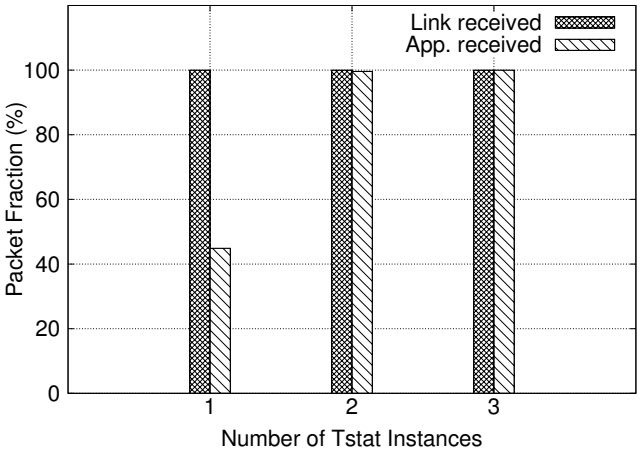

Fig. 7. Tstat and Linux socket: 10 Gbps traffic analysis with 300 Bytes average packet size

Figure 6 shows the results of the same test when the default Linux socket is replaced by PFQ with 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. In this case, however, fan–out does not accelerate the application performance. This, in fact, is the expected effect of fan–out and it is consistently observed as the internal lock–free implementation of the PFQ socket queue manages multi–core access contention efficiently.

VI. 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. As will be elaborated upon, in some cases the fan–out alone allows to scale–up 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*, is 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*, is 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, and will be represented as a fraction of the packet received at the socket level (*Link received*);
- *App. dropped*, is 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 [14] is a widely 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 7 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.

Figure 8 reports the results of the same experiment when the average packet size is set to 128 Bytes (and the corresponding average packet rate grows up to 8.2 Mpps). In this case, the use of fan–out allows two working instances of *Tstat* to effectively processes all of the packets received on the physical device. However, nearly 40% of the input packets turns out to be dropped at the network interface as the input traffic rate exceeds the potential capture rate of `TPACKET` socket. To further improve the performance of the application, packet fan–out can conveniently be combined with underlying socket

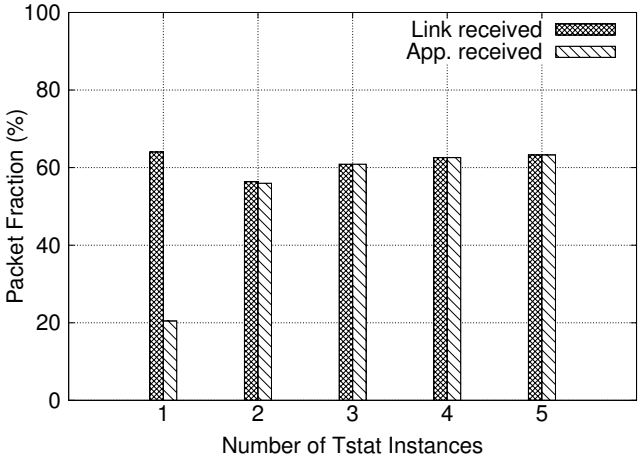

Fig. 8. Tstat and Linux socket: 10 Gbps traffic analysis with 128 Bytes average packet size

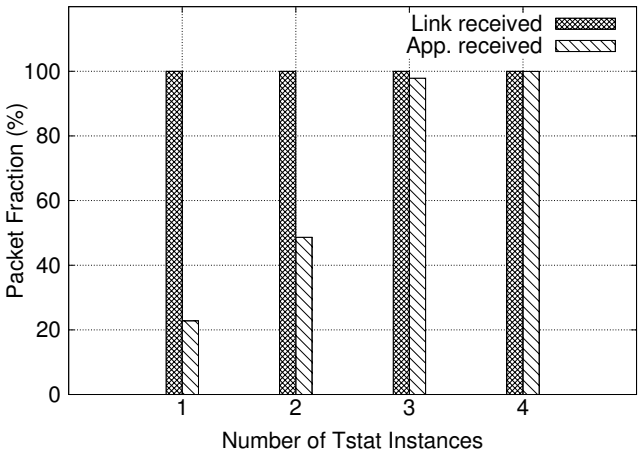

Fig. 9. Tstat and PFQ: 10 Gbps traffic analysis with 128 Bytes average packet size

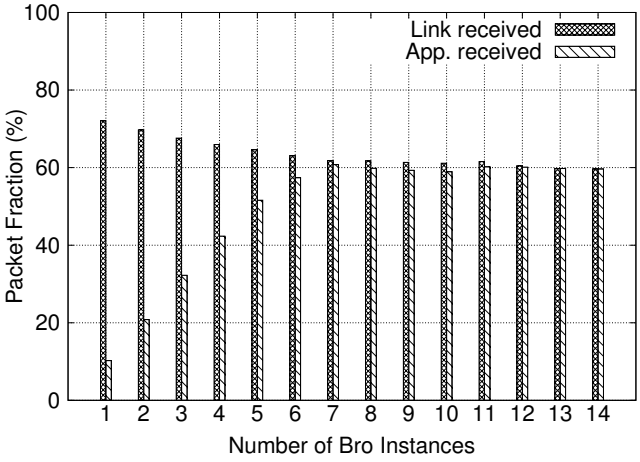

Fig. 10. Bro: real traffic analysis with standard Linux socket

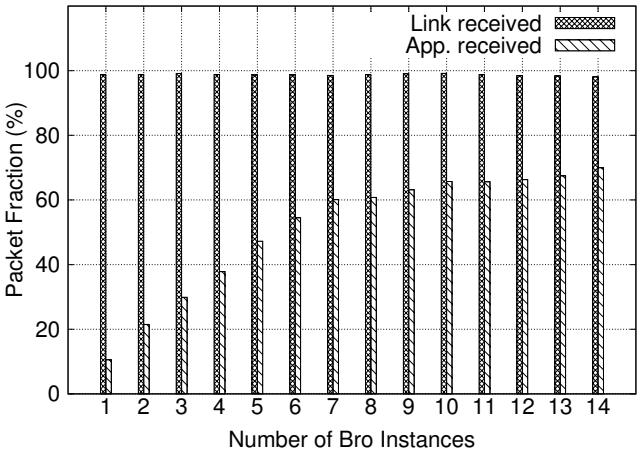

Fig. 11. Bro: real traffic analysis with PFQ

acceleration. Indeed, as shown in Figure 9, the use of PFQ allows to avoid packet drop at the lower level and packet fan–out allows three instances of *Tstat* to successfully process nearly all of the input traffic.

B. Bro

Analogous tests have been carried out to assess the performance of the *Bro* network security monitor [15] 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 syncronized by the *broctl* manager to provide a unified output.

So far, 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 without the need for extra plugins.

In the next experiments, a cluster of *Bro* nodes using the new `pcap` library is fed with a real packet trace played at 2.4 Mpps, corresponding to full 10 Gbps line speed.

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 10 shows the cluster performance when the standard Linux socket was used with two underlying capturing cores (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 relevant (up to 40%) and raises the need for socket acceleration.

Indeed, Figure 11 shows the results obtained when the

standard socket is replaced by PFQ under the same number (two) of capturing cores. The use of the accelerated socket dramatically reduces the packet drop rate at the interface up to negligible values. This 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 hyper–threading does not have the computational power of physical CPUs. Finally, notice that the number of physical cores of the PCs used in the experimental setup limited the maximum cluster cardinality to 14 nodes, as two of the overall 16 available cores were dedicated to underlying capturing/steering operations.

VII. 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 sockets.

Acknowledgment

This work has been partly supported by the EU project Behavioral Based Forwarding (BEBA, Project ID: 644122).

References

[1] Cisco Systems, "Cisco Visual Networking Index: Forecast and Methodology," June 2011. [Online]. Available: http://www.cisco.com

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

[3] 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

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

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

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

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

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