Portable packet processing modules for OS kernels [*]

Luigi Rizzo
Università di Pisa, Italy
rizzo@iet.unipi.it

This paper appears in IEEE Network, vol.28 issue 2, pg.6-11, March 2014

ABSTRACT

During the last fifteen years we have been involved in the design and development of some extremely popular pieces of open source software. Two of them, the dummynet network emulator and the netmap framework, are available as kernel components for popular operating systems, and are widely used in several research and commercial projects.

In this paper we will briefly describe the internals of the two systems, discuss the challenges in building kernel components that run on multiple Operating Systems, and analyse the difficulties in developing and maintaining open source software as part of one's academic activity.

1. INTRODUCTION

For a long time now we have designed, developed, distributed and supported several opensource software systems. Some of them have been developed as a standalone projects; others, including the two described here, were soon integrated in the FreeBSD Operating System and hence had to abide to the standards of the parent project.

Even for those systems directly related to our research work, a good part of the development effort has occurred well after the original research was completed, and has been supported mostly by personal interest and enjoyment in this activity.

Several of our systems, including the dummynet network emulator [2] and the netmap [13] high speed packet processing framework, have become extremely popular and widely used. Others, such as the erasure code in [11] and the PGMCC multicast congestion control scheme [14] have seen a slightly different fate, with a decline in popularity after some time.

In this paper we will present `dummynet` and `netmap`, which are kernel components supporting different Operating Systems. In addition to technical issues (features, performance) we will discuss the choices that (in our opinion) made them popular, and comment on how developing opensource software relates to academic activity.

2. DUMMYNET

The dummynet network emulator [2] saw the light in late 1995 when doing research on TCP congestion control. As it often happens in networking research, we needed to study the behaviour of a complex system, including applications, user libraries, in-kernel protocol implementations and device drivers, in a network environment that is not easily available or reproducible.

Two extreme approaches are used in these cases. One is recourse to full simulation, where every piece of the system (including traffic sources and sinks) is modeled so that their behaviour can be reproduced together with that of the network. The NS2 [1] simulator and similar tools include good models for physical layer and protocols. However application libraries, traffic sources and sinks normally must be (re)implemented by the user on a case by case basis.

The alternative is to try and reproduce the desired network conditions in a real testbed, sparing the modeling effort involved in simulation (often a monumental one, especially for the endpoints if one wants to reproduce them with reasonable fidelity). At the time, these experiments were often run borrowing computer accounts from fellow researchers at other institutions; more recently, platforms such as PlanetLab [5] have simplified the setup of the experiments. Either way, network conditions in these testbeds are hard to control or reproduce.

Dummynet is a network emulator, and its goal is to mix the good features of simulation (better reproducibility and no need to own/use a potentially expensive network infrastructure) and real experiments (no need to model all pieces of a system; ability to study black boxes that are part of the system under test).

Network emulators act on live traffic and actual protocol implementations. They simulate the effect of the communication links by intercepting traffic (either within the OS, or in the network itself), and delay/drop packets as if they were flowing through a communication channel with given features (bandwidth, propagation delay, queue size). These parameters are used to configure the object that emulates a communication link, called a "pipe" in dummynet's terminology (Figure 1).

2.1 Dummynet's operation

As shown in Figure 2, dummynet intercepts packets within the Operating System's kernel (at the IP or Ethernet layer, depending on the configuration) and inserts them in a queue which is part of the pipe. Each packet is then scheduled to emerge from the pipe after a delay

$$\Delta T = \frac{(Q_i + L_i)}{B} + t_{PD}$$

where Q_i is the queue occupation before packet i is enqueued, L_i is the length of the packet, B is the bandwidth

[*]The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n.287581 - OPENLAB.

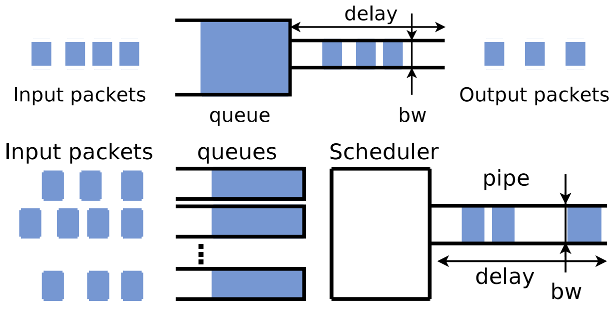

Figure 1: Top: a basic dummynet pipe and its parameters. Bottom: the emulated link can be driven by a packet scheduler serving multiple queues with different weights/priorities.

of the channel and t_{PD} is the propagation delay (length of the link divided by the speed of light in the medium).

The delay can be easily computed in the emulator, which then only needs to hold the packet in memory and schedule an event after ΔT to complete the delivery to the network card or to the rest of the protocol stack. This way, applications whose traffic is flowing through the pipe will experience an effective bandwidth and delay in the communication equal to the values configured for the pipe. This happens without any modification to the application or the OS.

Emulation also takes care of dropping packets when the queue attached to the pipe is full, same as it would occur on a real communication link. This is called congestion-related packet loss, and is one of the inputs to congestion control schemes such as the ones used in TCP. Dummynet can also emulate random packet drops, which is useful to test the response of a system to losses not related to congestion.

2.2 Traffic selection

Reproducing the behaviour of a link is only part of the operation of an emulator. Equally important is the selection of traffic that should be subject to emulation: a PC/workstation normally sends and receives plenty of traffic that is not part of the experiment, and should not be affected by the emulator. Most ad-hoc emulators are extremely inflexible in their filtering abilities, and tend to force researchers to adapt the experiment to the tool, rather than the other way around.

In dummynet, traffic selection (see Figure 2) is done by the system's firewall, `ipfw`, which has been extended to provide a simple yet powerful configuration mechanism. We can define multiple pipes with independent configurations, and use `ipfw`'s packet matching options to send traffic to specific pipes. With relatively simple and intuitive configuration commands we can model asymmetric links, multiple paths, and pass traffic through a sequence of queues/pipes, allowing the emulation of complex network topologies with multiple bottlenecks and inputs/outputs.

As an example of the simplicity of configuration, the commands below create two pipes to model an asymmetric link,

```
ipfw pipe 10 config bw 256 Kbit/s delay 20ms
ipfw pipe 11 config bw   4 Mbit/s delay 5ms
```

and another couple of ipfw rules suffice to pass traffic of interest (e.g. for a given port range) to one or the other pipe, depending on the direction

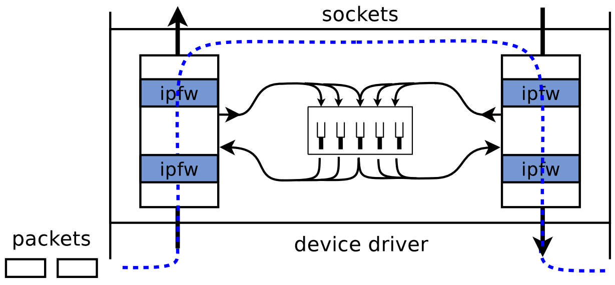

Figure 2: The placement of the emulator within the system. The ipfw classifier selects traffic for the various pipes, while multiple pipes emulate queues, schedulers and links.

```
ipfw add 100 pipe 10 out dst-port 2000-3000
ipfw add 200 pipe 11 in  src-port 2000-3000
```

and finally, one more rule is used to bypass the emulator for all other traffic

```
ipfw add 300 allow ip from any to any
```

2.3 Traffic scheduling

When we started some research on packet scheduling [4], dummynet became a useful tool for evaluating the performance of our algorithms. To help our research, we added support for configurable packet schedulers (in the form of loadable kernel modules) in front of a pipe, and extended ipfw so that pipes, schedulers and traffic classes could be created very easily.

The configuration scheme was carefully designed so that a small amount of commands could produce very complex and flexible settings. As an example, below we show how to build a traffic shaper that gives independent 1 Mbit/s pipes to every /24 subnet connected to it, and implements fair queueing within each subnet with two different traffic classes per host.

We start by creating a set of pipes with the desired bandwidth (the parameter `mask src-ip 0xffffff00` creates one pipe for each /24 source subnet), each driven by a QFQ [4] scheduler.

```
ipfw pipe 5 config sched qfq mask src-ip 0xffffff00 bw 1 Mbit/s
```

We then define two traffic classes ("queues" in dummynet terminology) with different weights (i.e., receiving different fractions of the available bandwidth). Again, the `mask ...` parameter causes the creation of multiple queues, in this case one per source IP. Queues are connected to the scheduler/pipes according to the masks: eventually, each pipe (representing a /24 network) will be driven by up to two queues (with different weights) per source IP.

```
ipfw queue 1 config pipe 5 weight 10 mask src-ip 0xffffffff
ipfw queue 2 config pipe 5 weight  4 mask src-ip 0xffffffff
```

The final step of the configuration is to send traffic to one or the other (set of) queues as desired. In this example we send ssh and DNS to the queue with higher weight, and all remaining traffic to another one with a lower weight.

```
ipfw add 100 queue 1 in dst-port 22,53
ipfw add 200 queue 2 in dst-port any
```

2.4 Additional features

As network technologies evolved, the basic pipe model of dummynet was further extended to model additional link features, such as the effect of multiple paths between endpoints, or wireless links, where the effect of link arbitration and competing stations on the throughput and delay is not adequately represented by the basic pipe model. The solution we adopted for dummynet was simple but effective: each pipe can be further configured with an empirical "delay profile", which determines, in a probabilistic way, the time that the channel is unavailable between packet transmissions to arbitrate requests from competing stations.

While developing dummynet extensions, we also refactored the `ipfw` classifier to make it more efficient and easier to extend. In addition to the basic selection options based on individual packet fields, we and others also introduced a number of options to deal with metadata, or efficiently handle sets of addresses and ports. To date, the combined manual page for ipfw and dummynet is over 2000 lines of text, witnessing the large set of features that have been made available over time.

2.5 Performance and limitations

Knowing the limitations of a live emulation system is fundamental to perform significant experiments. In our case, since the emulation modifies the delivery time of each packet, the two important parameters are the granularity of timers, and the processing time spent on each packet.

The granularity issue comes from the way the system operates. Packets need to be stored and dispatched at a later time, and because scheduling an event to dispatch the packet is moderately expensive (and it was even more so in the systems of 15 years ago), dummynet rounds all times to a multiple of the period of the system's timer (default to 1 ms, but configurable; we have successfully run dummynet on modern CPUs with 25..50 μs granularity). This limits the number of events to schedule, and largely reduces the cost of emulation.

The rounding of times does not affect the precision of emulation (bandwidth can be configured with a resolution of 1 bit/s), but may cause some burstiness at high data rates and/or with small packets. As an example, a 1500-byte packet on 1 Gbit/s links consumes approximately 12 μs, so we can expect some burstiness compared to a real 1 Gbit/s link. Note that it is not obvious that the result will be different for the receiving application, as normally network cards limit the maximum interrupt rates and this also causes the delivery of packets to the software in similar bursts.

In terms of performance and scalability, packet processing costs in dummynet are either constant or logarithmic in the number of active pipes. We also modified extensively the ipfw classifier to reduce the algorithmic complexity (and absolute cost) of traffic selection. Detailed measurements are reported in [3, Table 1]; as a reference, a modern system can easily sustain around 500 K packets per second (pps) through a pipe and, as discussed in Section 3.3, dummynet and ipfw are not the performance bottleneck.

2.6 Porting to different platforms

Dummynet was originally developed on FreeBSD (and imported in OSX together with the rest of FreeBSD). Being part of a widely used OS kernel means that the code had to abide to strict constraints on quality, performance, backward compatibility, even style. All this was significantly time consuming at the beginnig, but definitely helped to keep the code alive and maintainable for over 15 years.

Since the very beginning, we received numerous requests to port dummynet to other OSes (Linux and Windows; OSX became automatically supported when Apple decided to use the FreeBSD kernel as the basis for its operating system). However these requests were not backed by any type of support to help the development work. The Linux (and later Windows) ports were only possible a few years ago thanks to an EU-funded research project called ONELAB2. Within ONELAB2, we extended the PlanetLab testbed with emulation capabilities based on dummynet, and this prompted the development of a Linux version of dummynet, soon followed up by a Windows version.

Adapting a kernel component to different OSes is an interesting challenge, because OSes differ in internal data representations and software interfaces. Fortunately, dummynet and ipfw only interacts with a limited set of the kernel's interfaces and data structures (mostly, packet representation: mbufs on FreeBSD, skbufs on Linux, NDISPacket in Windows). This enabled us to use a technique that proved useful in other subsequent projects: we built wrappers to map FreeBSD data structures and kernel APIs into equivalent functions for the target operating systems. As a result, subsequent enhancements to dummynet and ipfw were immediately available with little or no effort to all other supported OSes.

2.7 Distribution and support

To date, dummynet is widely used both in research (over a thousand citations according to Google Scholar) and in operations. As an example, it is the emulation engine used by the Emulab [19] testbed, and a lot of large and small IT companies use it in their testing labs.

Features and flexibility helped but were not sufficient to achieve this popularity. We needed to gain user confidence (especially critical when dealing with kernel components), minimize deployment costs (as an example, this is an issue if the tool is not available for your platform, or requires time-consuming installations), and avoid a steep learning curve (which could happen when the tool is complex to use even for simple tasks).

The first two issues were addressed by distributing [9] a fully bootable, standalone image of a complete FreeBSD system which could be run on a PC without any persistent installation on the hard disk. A build system called "picobsd" let us store all the required components (including an ssh client and server) within a single floppy disk image (1.44 MBytes). This way, one could quickly repurpose an inexpensive PC as a transparent bridge acting as a flexible network emulator, something also called "bump in the wire" (Figure 3). This was back in 1998-1999, before the concept of "virtual appliances" became widespread, and similar systems were either expensive appliances or available on high end workstations.

The integration of dummynet within standard distributions of FreeBSD (and later of OSX) also contributed to its popularity, because users would find the tool already available on their system without any need to modify the kernel.

Regarding the learning experience, we tried hard to make sure that the basic features in dummynet were usable without having to read a long manual. A two-line configuration

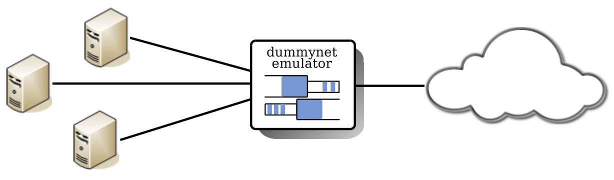

Figure 3: A picobsd emulator acting as a "bump in the wire". The emulator acts as a transparent bridge, so there is no change required in the host or network configuration to add emulation within a network. Emulab and many other testbeds use exactly this principle, using dummynet to emulate a complex network with the desired features.

is all is needed to have a useful emulation environment, and the short examples presented earlier can be considered complex configurations.

It should be noted that while these decisions had a significant impact on making the system popular, they also placed a significant burden on us. Especially in the early years, support and performance and stability fixes sometimes required massive work both on the emulator and on the host operating system.

3. NETMAP

The second system we discuss is the netmap framework [12, 13], a recent effort to make commodity operating systems cope with the ever increasing speed of network interfaces.

The problem addressed by netmap is that network protocol stacks, designed 20-30 years ago, have a hard time keeping up with packet rates that have increased by 3-4 orders of magnitude in the meantime. A 10 Gbit/s interface may have to deal with up to 14.88 million packets per second (Mpps), whereas the typical operating system has normally troubles handling more than 1-2 Mpps per core. In contrast with this speed mismatch, there is an increasing demand for high speed software packet processing solutions, triggered by the growing popularity of software defined networking (SDN) and virtual machines (VMs).

As a partial remedy to the problem, modern network interface cards (NICs) expose multiple transmit and receive queues, to achieve better scalability with multicore CPUs. On the software side, some vendors addressed the problem with user-space protocol implementations that bypass the operating system and directly access the hardware (Solarflare's OpenOnLoad [18], Intel's DPDK [8]). Similar approaches, restricted to a single card or OS, have been implemented by Deri [6] and the packetshader I/O library [7].

Considering that all existing solutions had significant hardware or software restrictions, we explored the problem of raw packet I/O, trying to design an architecture as much as possible independent from the specific device or OS, yet able to deal with line rate at 10 Gbit/s without consuming an inordinate amount of CPU cycles, and exploiting the protection and synchronization mechanisms that OSes make available.

The main cost factors in packet processing include system calls, memory management (allocations and copies), and the many software layers that a packet has to traverse in order for the kernel to implement the various features it provides.

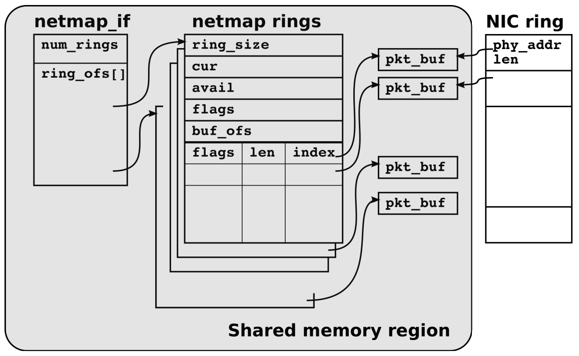

Figure 4: Data structures used by netmap

Our design thus tried to remove or amortize many of these operations in order to improve performance.

3.1 Architecture

From a user's point of view, netmap introduces a new API to send and receive raw packets from userspace (Figure 4). Applications see a ring of buffer descriptors and packet buffers, shared with the kernel, and synchronize with the latter through system calls operating on a file descriptor. Data transfers occur in batches, whose size is chosen by the user; this amortizes system call costs in a very nice way (as the load grows, batches tend to be larger, and the system becomes more efficient). Data copies are eliminated because buffers are directly visible to the network interface, making true zero-copy operation possible (one should not forget, however, that a significant amount of memory bandwidth is still consumed by just reading data, or having the NIC read/write packets to buffers in main memory). In netmap, memory allocations (a major cost component in network I/O) only occur when the file descriptor is opened and not on every packet.

We provide non-blocking `ioctl()`'s, as well as blocking calls (`select()` or `poll()`) that return when there are buffers available.

Non blocking I/O is useful for applications that want minimal interference (and delay) from the operating system in doing I/O. It is also a common choice for many "OS bypass" systems, as it saves the effort of implementing support to synchronize device I/O with the operating system. The downside of this approach is that it consumes all of the CPU time spinning for events.

The blocking file descriptor supported by netmap is a rather unique feature among high performance network I/O frameworks, and one that permits very efficient resource usage. Also, this design choice proves incredibly useful when it comes to adapt existing applications to the new API. In fact, many packet processing systems already rely on select-able file descriptors, handled within event loops, to determine when I/O can be performed. Thanks to the same interface, we were able to build an extremely simple libpcap wrapper on top of the netmap API, and this let us run unmodified binary applications on our library with sometimes large performance improvements.

3.2 Implementation

Netmap is implemented as a kernel module, and is made of two parts: generic code handles the user API (ioctls, memory mapping, synchronization), while device-specific backends interact with the hardware. These backends extend the existing device drivers by adding and a small number of functions to send and receive packets using the netmap-specific data representation.

Once again, a big difference between netmap and other systems with a similar goal is that we reuse most of the existing device drivers' code. Our extensions are limited to 4-500 lines per driver (compared to the 3..10 K lines of code that typically make up a device driver), and mostly affect the transmit and receive routines, which are generally easy to understand and debug.

To date, netmap is available for FreeBSD (in which is part of standard distributions) and Linux (as an external kernel module), and supports 7 different cards from four manufacturers (Intel, Realtek, nvidia, and partly Mellanox).

3.3 Performance

For the tasks it has been designed for (generic packet processing, routers, traffic sources, monitors) netmap achieves fantastic speedups, 10..40 times faster than native OS functions for the same purpose. Detailed performance results are presented in [13]; the most impressive result is the ability to send or receive minimum sized packets at line rate on a 10 Gbit/s interface (14.88 Mpps) using one core at less than 1 GHz. This matches the efficiency of the best state of the art solutions, while at the same time providing a much higher flexibility of use (both in terms of user interface and HW/OS independency).

It is worth mentioning that as a proof of concept we modified ipfw and dummynet to run in user space on top of netmap (using an approach similar to the one used for the Linux and Windows port), reaching a filtering speed of over 6 Mpps for the firewall, and over 2 Mpps for packets going through dummynet.

The netmap API supports an easy and efficient emulation of the very popular PCAP API, which means that programs using that API can run on top of netmap without even recompiling and with huge performance improvements – between 4 and 10 times, depending on the application [15].

3.4 Extensions

Since its initial version, we have extended netmap in several ways. A port from the original FreeBSD version to Linux was done with techniques similar to those used for dummynet. We used the netmap API to design and build a high performance virtual ethernet switch called VALE [16], meant as an interconnect between virtual machines, but also useful to test netmap-based applications without requiring high speed cards. VALE can forward up to 20 Mpps per core with small packets, with a total throughput only limited by the memory bandwidth of the system (current measurements indicate up to 120 Gbit/s, but the value depends on how the memory bus is used by other parts of the system). Finally, we worked on the QEMU/KVM hypervisor to exploit the speed made available by VALE. Our current prototype [17] supports over 5 Mpps between two virtual machines, several times larger than the original version.

3.5 Distribution and support

Despite being only two years old, netmap is gaining a lot of interest in the research community, and also in the commercial world (basing on interactions we had with companies and developers. Our code is distributed with a two-clause BSD license, so companies using netmap in a product are in no obligation of advertising its use in documentation, or contributing back modifications).

The strategy to improve popularity was the same used for dummynet: we imported the code in FreeBSD distributions, and made available [10] pre-built system images with a few sample programs to ease experimenting with the tool. In fact, our netmap image became a popular replacement for expensive 10 Gbit/s traffic sources and sinks. Full availability of source code compatible with many different Linux distributions also helped increase the use of our framework. Once again, especially in the initial period, there was a significant support effort to answer many questions and address bugs and compatibility problems.

4. ACADEMIC RECOGNITION OF OPEN-SOURCE WORK

Both dummynet and netmap have been extremely successful and widely used. We attribute at least part of this success to the extensive effort we put in refining and supporting our software, making it easy to use and reliable. Our code is normally distributed under a BSD license, and some of it has been included even in commercial, closed source products.

As discussed in Sections 2.7 and 3.5, the move from a proof-of-concept to a "production quality" system required very specific and time-consuming actions. This is hardly surprising: making software (and kernel components in particular) useful requires quality standards much closer to commercial products than to research prototypes. The amount of work needed for "productization" (full development, testing, documentation, distribution, support) is massive and very different from what is needed for a simplified evaluation of research ideas.

In terms of personal satisfaction, building these (and other) open source systems and tools for networking research has been extremely rewarding, and we feel that our work has and will contribute to support further research.

It is however unfortunate, and very frustrating, that in our experience (both as authors and as reviewers for top conferences in the area), the networking academic community does not seem to give adequate recognition to this type of activity. Papers describing software tools or protocol/application implementations are often rejected, on the grounds that most of their contributions are on engineering the system or combining known ideas and algorithms – and as such they do not qualify as research. Funding agencies, both public and private, often use similar metrics (and perhaps the same reviewers!), preferring more ambitious but often more risky and abstract research proposals to those aimed at developing tools (which include most opensource work). In fact, the vast majority of support for open source systems (such as Linux, FreeBSD, Qemu/KVM) comes from commercial companies rather than from research funding.

Ironically, we often received (or read, related to other papers) rejection reviews containing the statement "... but I am very interested to use this tool for my research." The latter does indeed happen, suggesting that such allegedly "non-research" contributions are actually valuable, do contribute to the achievement of further research results, and their existence avoids replication of efforts and makes for a more efficient use of researchers' time and skills.

We strongly believe that the networking research community and funding agencies should give more adequate recognition to experimental/practical work that results in the development of (good quality) tools and systems. Not doing so has unhealthy consequences, including depriving researchers of useful means for validating their results, underestimating or ignoring fundamental engineering issues, and generally increasing the disconnect between academia and industry, making them evolve on diverging paths.

5. REFERENCES

[1] The ns-2 Network Simulator. http://nsnam.isi.edu/nsnam/.

[2] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, Apr. 2010.

[3] M. Carbone and L. Rizzo. An emulation tool for planetlab. *Computer Communications*, 34(16):1980–1990, Oct. 2011.

[4] F. Checconi, P. Valente, and L. Rizzo. QFQ: Efficient Packet Scheduling with Tight Bandwidth Distribution Guarantees. *IEEE/ACM Transactions on Networking*, 21(3), 2013.

[5] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[6] L. Deri. PFRING DNA page. http://www.ntop.org/products/pf_ring/dna/.

[7] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[8] Intel. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378*, 2012.

[9] L. Rizzo. Dummynet home page. http://info.iet.unipi.it/∼luigi/dummynet/.

[10] L. Rizzo. The netmap project. http://info.iet.unipi.it/∼luigi/netmap/.

[11] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[12] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC'12*, Boston, MA. USENIX Association, 2012.

[13] L. Rizzo. Revisiting network I/O APIs: the netmap framework. *Communications of the ACM*, 55(3):45–51, Mar. 2012.

[14] L. Rizzo. pgmcc: a tcp-friendly single-rate multicast congestion control scheme. In *ACM SIGCOMM 2000.* ACM, Stockolm, Aug-Sep. 2000.

[15] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Infocom 2012.* IEEE, Orlando, FL, March 2012.

[16] L. Rizzo and G. Lettieri. VALE, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72. ACM, Nice, Dec. 2012.

[17] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up Packet I/O in Virtual Machines. In *ANCS 2013.* IEEE, San Jose, Oct.2013.

[18] Solarflare. Openonload. *http://www.openonload.org/*, 2008.

[19] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.