# netmap: framework for very fast access to network devices

Matteo Landi
Dip. di Ingegneria dell'Informazione
Università di Pisa, Italy
matteo@matteolandi.net

## ABSTRACT

The increasing availability of high speed network adapters at very low costs, leads to the possibility of creating high performance network applications (software switches, traffic monitors and generators, firewalls, etc.) with the utilization of commodity hardware. However, in order to exploit the power of such devices, OS kernels need to be modified, casting away all those heavy operations thought for a general purpose usage. In this work we present **netmap**, a system that integrates the strengths of existing proposals and addresses their weaknesses. First an abstraction layer has been implemented on top of network adapter kernel structures. Then, these wrappers have been exposed to userspace by means of memory mapping, so that programs could directly manipulate packets at application level and synchronize with Kernel only when needed. With **netmap** we have been able to saturate the link capacity on both 1 Gbit and 10 Gbit network adapters. The reasons behind such increase of performance are to be found first and foremost in the preallocation of *packet buffers*, then in the reduced utilization of system-calls (possibly amortized over packet batches), and finally in the minimized overhead due to encapsulation and metadata management.

## 1. INTRODUCTION

One thing in common between systems such as software routers, switches, intrusion detection systems and monitors, is the ability to move packets as quickly as possible between the wire and the application.

There are three are the main components involved in such task, namely: network interface controllers (*NIC*s), device drivers, and the Operating System (*OS*). *NIC*s usually manage network packets with the aid of circular queues (*ring*s), containing the so called *packet descriptors*; each descriptor is a container in which device drivers store information about network packets, such as the physical address of the *packet buffer*, its length and some control flags. *OS* on the other hand keeps track of network packets by means of *shadow* copies of *NIC*s data structures in order to store useful information such as packet dimension, sender/receiver addresses or either control flags handling packet fragmentation.

However, depending on the needs it is possible for dedicated appliances to remove unnecessary software layers and address the problem by taking direct control of the hardware. This way researchers and developers succeeded in taking full advantage of the hardware and send millions of packets per second using modified Click Drivers [1, 2] or either exporting packet buffers to userspace [3]. Moreover, if on the one hand giving applications direct access to the hardware would enable the possibility to leverage the power of the network hardware, on the other makes the system vulnerable against potential malicious applications.

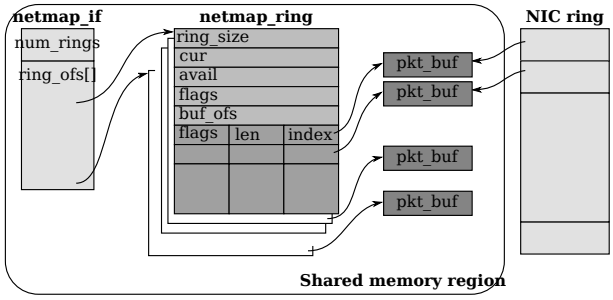


**Figure 1: netmap structures implementing the hardware independent abstraction layer (*netmap interface* and *netmap rings* shared with applications, and *packet buffers* shared with both applications and *NIC*s).**

The framework **netmap** [4] presented in this work, enables userspace applications to process (and possibly forward) network packets at wire-speed preserving the safety and the convenience of a rich and portable execution environment, as conventional *OS*s offer. Even if applications in need of sending and receiving packets at very high speeds are asked to be re-implemented based on the native **netmap** API, it is possible for existing applications to exploit the power of the framework thanks to wrapper libraries (e.g. mapping `libpcap` calls onto **netmap** ones).

## 2. ARCHITECTURE

Applications in need of putting a certain interface into **netmap** mode, would open the file `/dev/netmap` and issue a special `ioctl()`; as a direct consequence of this *registration*, *NIC*s get partially disconnected from the stack, for that standard `ioctl()` commands used to configure and modify interface options are still operational on **netmap** file descriptors. Packets on the other hand are no more exchanged with the host stack by mean of the standard *socket* interface, but rather are stored into pre-allocated buffers made available to applications through the system call `mmap()` (Fig. 1). This shared memory region is also used to store structures called *netmap rings* which represent an abstraction layer enabling applications to access network packets in a hardware independent way; moreover, particular attention has been

spent to minimize the amount of information stored inside these objects so that to reduce their management overhead (those structures contain indeed a pointer from which to start reading or writing packets, the number of available slots, and for each slot, the size of the payload, the index of the buffer, and a couple of status fields).

## 2.1 Multi-queue NICs

It is becoming more frequent to see high speed network adapters succeeding in sending millions of packets per second making use of multiple hardware queues so that multiple CPU cores could be assigned to each of them independently. **netmap** has been thought to benefit of this feature too, in particular to each network interface is associated a number of *netmap rings* equal to the number of hardware queues actually in use by the driver: this way it is possible to use different file descriptors to process different adapter queues in a complete independent way without any need of synchronization or any sort of locking mechanisms.

# 3. SYSTEM SAFETY

It is true that **netmap** enables applications to access the internal structures of network devices; but it's even more true that applications are allowed to modify only the content of received/transmitted packets or *packet buffers* indices. This way, device drivers are always guaranteed to cope with valid virtual and physical addresses only, hence malicious applications are unable to crash the whole system.

## 3.1 Packet processing

Applications interested in packet reception should first of all wait for packet availability: this is achieved, depending on the applications, by issuing a non-blocking `ioctl(fd, NIOCSYNCRX)` or either a blocking `poll(...)` call. On the return of these system calls, applications are guaranteed that the status of *netmap rings* is matching the one of the hardware ones, hence `avail` packets starting from the `cur`-th ring slot are ready to be processed.

On the transmit side things are not that different: in particular, applications will issue either a `ioctl(fd, NIOCSYNCTX)` or a `poll()` in order to find if there is room for at least one packet to be filled; on the return of such calls, `avail` packets are ready to be filled by applications, starting from the `cur`-th slot of the ring.

Finally, values of rings modified also by userspace applications (i.e. `cur` and `avail`) will be used in subsequent system call to tell the kernel which buffers have been consumed.

# 4. PERFORMANCE ANALYSIS

We were able to test the performance of the implemented framework inside different scenarios, varying for example the model of the network adapter, the clock speed of the CPUs, and also the number of active CPU cores; moreover, we developed a couple of *ad hoc* applications (common network traffic analysis applications) able to leverage the new network machinery.

Using a dual port 10 Gbit/s card based on the Intel 82599 chip, mounted on a system equipped with an Intel Core i7-870 Processor, we were able to send minimal size packets (64 bytes) at peak rates of 14.88 millions of packets per second (Fig. 2). For additional information a more detailed performance analysis is presented in [4].

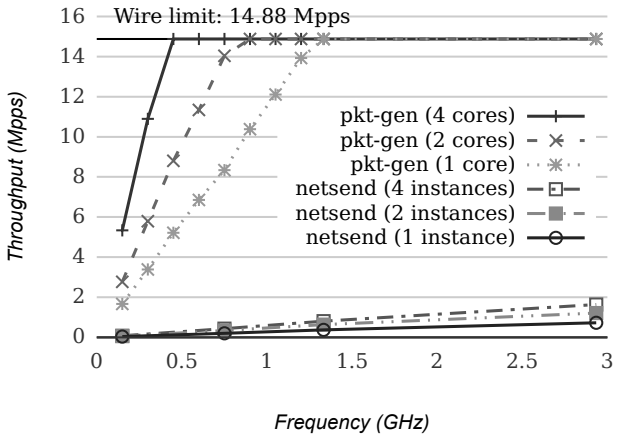


**Figure 2: TX throughput comparison between a netmap-based packet generator and a standard one.**

# 5. STATUS AND FUTURE WORK

The current prototype of **netmap**, developed on FreeBSD, consists of about 2000 lines of code for device functions (`ioctl()`, `select()`/`poll()`) and driver support, plus individual device driver modifications (mostly mechanical, about 500 lines each), to interact with the *netmap rings*. Moreover, a `pcap`-like library has been developed to make existent applications (e.g. Click and Open vSwitch) *transparently* benefit of the new framework [5].

Even if few standard drivers have already been modified to add **netmap** support (`ixgbe`, `em`, `re`, `rl`), we hope to spread the number of supported *NIC*s as much as possible; moreover, ongoing work is trying to implement new features to the framework such as enabling applications to modify a number of rings greater than the actual number of hardware ones supported by network adapters (in order to let multiple applications to work with the same network adapter despite the number of hardware queues).

# 6. REFERENCES


[1] E.Kohler, R.Morris, B.Chen, J.Jannotti, M.F.Kaashoek, The Click modular router, ACM TOCS, vol.18, pp.263–297 2000

[2] M.Dobrescu, N.Egi, K.Argyraki, B.G.Chun, K.Fall, G.Iannaccone, A.Knies, M.Manesh, S.Ratnasamy, RouteBricks: Exploiting parallelism to scale software routers, ACM SOSP, 2009

[3] S.Han, K.Jang, K.Park, S.Moon, PacketShader: a GPU-accelerated software router, ACM SIGCOMM, 2010

[4] L.Rizzo, netmap: fast and safe access to network adapters for user programs, Technical report, http://info.iet.unipi.it/∼luigi/netmap/rizzo-ancs.pdf, Università di Pisa, Italy, 2011

[5] L.Rizzo, M.Carbone, G.Catalli, Transparent acceleration of software packet forwarding using netmap, Technical report, http://info.iet.unipi.it/∼luigi/20110729-rizzo-infocom.pdf, Università di Pisa, Italy, 2011