Moving packets quickly between the wire and the application is a must for systems such as software routers, switches, firewalls, traffic generators and monitors. But at 10 Gbit/s line rate equals to 14.88 Mpps per port, or 67.2 ns per packet. Sure, there are a few custom solutions [1, 2, 3, 4, 5] that run really fast. But how do we achieve such speeds (and remain compatible with applications written in the past 20 years) on general purpose OS designed when “fast” was 2-3 orders of magnitude lower ?

netmap [6] addresses the problem by making the datapath between the wire and userspace applications as fast as possible, but otherwise leaving unchanged the rest of the OS, and most/all user APIs. As a result we achieved 10..20 times faster I/O rates for the applications as fast as possible, but otherwise leav ing unchanged the rest of the OS, and scale slightly better to 1..2 Mpps for in-kernel apps, 0.5 Mpps for userspace apps, up to 4 Mpps, and has with very good scalability on clock frequency and number of cores. With netmap we do line rate packet generation at 10 Gbit/s (14.88 Mpps) with a single core running at 2 GHz (and low CPU occupation at higher clock rates). Packet reception is equally fast.

As a result, current per-core performance on standard OS is 0.5 Mpps for userspace apps, up to 1..2 Mpps for in-kernel apps, and poor scalability with number of cores. Custom systems (in-kernel Click [1]) reach about 4 Mpps, and scale slightly better with cores. The NIC would be able to manage circular lists of buffers with little/no CPU intervention, but the OS does not make good use of these features.

netmap’s key ideas
- a shadow copy of the NIC’s ring (netmap ring) supports batching of requests and removes the need for mbufs/skbufs;
- efficient synchronization using poll();
- carefully designed API, event loops need only one syscall per iteration;
- full support for multicore and multiqueue NICs through setaffinity();
-通告 fetch, send, receive
- expensive software modifications are minimized as follows:
  - device independent API, does not rely on specific hw features;
  - minimal, mostly mechanical modifications to existing device drivers;
  - packets from/to the host stack can still use the NIC;
  - we provide an efficient libpcap emulation library on top of the native API.

Most of these ideas above have been proposed before, but separately. None of the existing systems to date (though [4] comes close) puts all these features together into a high performance and general purpose framework for packet I/O from userspace. A well engineered architecture is much more than a collection of parts.

Using netmap (native API)
Threads open /dev/netmap and issue an ioctl() to switch the NIC to “netmap” mode, disconnecting the datapath from the host stack. Data packets and netmap rings are in an mmap()d region with well defined ownership, so that lock free access is possible. Netmap rings are updated by poll() or ioctl(), and their content is validated by the kernel so a faulty program cannot crash the system. File descriptors and netmap rings can be associated to individual rings (and cores). A thread can manage multiple interfaces and do zero-copy forwarding to other interfaces or to the host stack.

Sample code for a packet generator:

```
Sample code for a packet generator:

```

Expensive software modifications are minimized as follows:
- device independent API, does not rely on specific hw features;
- minimal, mostly mechanical modifications to existing device drivers;
- packets from/to the host stack can still use the NIC;
- we provide an efficient libpcap emulation library on top of the native API.

Most of these ideas above have been proposed before, but separately. None of the existing systems to date (though [4] comes close) puts all these features together into a high performance and general purpose framework for packet I/O from userspace. A well engineered architecture is much more than a collection of parts.

Using netmap (native API)
Threads open /dev/netmap and issue an ioctl() to switch the NIC to “netmap” mode, disconnecting the datapath from the host stack. Data packets and netmap rings are in a mmap()d region with well defined ownership, so that lock free access is possible. Netmap rings are updated by poll() or ioctl(), and their content is validated by the kernel so a faulty program cannot crash the system. File descriptors and netmap rings can be associated to individual rings (and cores). A thread can manage multiple interfaces and do zero-copy forwarding to other interfaces or to the host stack.

```

Expensive software modifications are minimized as follows:
- device independent API, does not rely on specific hw features;
- minimal, mostly mechanical modifications to existing device drivers;
- packets from/to the host stack can still use the NIC;
- we provide an efficient libpcap emulation library on top of the native API.

Most of these ideas above have been proposed before, but separately. None of the existing systems to date (though [4] comes close) puts all these features together into a high performance and general purpose framework for packet I/O from userspace. A well engineered architecture is much more than a collection of parts.

Using netmap (native API)
Threads open /dev/netmap and issue an ioctl() to switch the NIC to “netmap” mode, disconnecting the datapath from the host stack. Data packets and netmap rings are in a mmap()d region with well defined ownership, so that lock free access is possible. Netmap rings are updated by poll() or ioctl(), and their content is validated by the kernel so a faulty program cannot crash the system. File descriptors and netmap rings can be associated to individual rings (and cores). A thread can manage multiple interfaces and do zero-copy forwarding to other interfaces or to the host stack.

```

Expensive software modifications are minimized as follows:
- device independent API, does not rely on specific hw features;
- minimal, mostly mechanical modifications to existing device drivers;
- packets from/to the host stack can still use the NIC;
- we provide an efficient libpcap emulation library on top of the native API.

Most of these ideas above have been proposed before, but separately. None of the existing systems to date (though [4] comes close) puts all these features together into a high performance and general purpose framework for packet I/O from userspace. A well engineered architecture is much more than a collection of parts.

Using netmap (native API)
Threads open /dev/netmap and issue an ioctl() to switch the NIC to “netmap” mode, disconnecting the datapath from the host stack. Data packets and netmap rings are in a mmap()d region with well defined ownership, so that lock free access is possible. Netmap rings are updated by poll() or ioctl(), and their content is validated by the kernel so a faulty program cannot crash the system. File descriptors and netmap rings can be associated to individual rings (and cores). A thread can manage multiple interfaces and do zero-copy forwarding to other interfaces or to the host stack.

```

Expensive software modifications are minimized as follows:
- device independent API, does not rely on specific hw features;
- minimal, mostly mechanical modifications to existing device drivers;
- packets from/to the host stack can still use the NIC;
- we provide an efficient libpcap emulation library on top of the native API.

Most of these ideas above have been proposed before, but separately. None of the existing systems to date (though [4] comes close) puts all these features together into a high performance and general purpose framework for packet I/O from userspace. A well engineered architecture is much more than a collection of parts.