NAME

netmap - a framework for fast packet I/O

SYNOPSIS

device netmap

DESCRIPTION

netmap is a framework for extremely fast and efficient packet I/O for userspace and kernel clients, and for Virtual Machines. It runs on FreeBSD, Linux and some versions of Windows, and supports a variety of **netmap ports**, including

physical NIC ports
to access individual queues of network interfaces;

host ports
to inject packets into the host stack;

VALE ports
implementing a very fast and modular in-kernel software switch/dataplane;

netmap pipes
a shared memory packet transport channel;

netmap monitors
a mechanism similar to bpf(4) to capture traffic

All these **netmap ports** are accessed interchangeably with the same API, and are at least one order of magnitude faster than standard OS mechanisms (sockets, bpf, tun/tap interfaces, native switches, pipes). With suitably fast hardware (NICs, PCIe buses, CPUs), packet I/O using **netmap** on supported NICs reaches 14.88 million packets per second (Mpps) with much less than one core on 10 Gbit/s NICs; 35-40 Mpps on 40 Gbit/s NICs (limited by the hardware); about 20 Mpps per core for VALE ports; and over 100 Mpps for **netmap pipes**. NICs without native **netmap** support can still use the API in emulated mode, which uses unmodified device drivers and is 3-5 times faster than bpf(4) or raw sockets.

Userspace clients can dynamically switch NICs into **netmap** mode and send and receive raw packets through memory mapped buffers. Similarly, **VALE** switch instances and ports, **netmap pipes** and **netmap monitors** can be created dynamically, providing high speed packet I/O between processes, virtual machines, NICs and the host stack.

netmap supports both non-blocking I/O through ioctl(2), synchronization and blocking I/O through a file descriptor and standard OS mechanisms such as select(2), poll(2), kqueue(2) and epoll(7). All types of **netmap ports** and the **VALE switch** are implemented by a single kernel module, which also emulates the **netmap** API over standard drivers. For best performance, **netmap** requires native support in device drivers. A list of such devices is at the end of this document.

In the rest of this (long) manual page we document various aspects of the **netmap** and **VALE** architecture, features and usage.

ARCHITECTURE

netmap supports raw packet I/O through a *port*, which can be connected to a physical interface (*NIC*), to the host stack, or to a **VALE** switch. Ports use preallocated circular queues of buffers (*rings*) residing in an mmapped region. There is one ring for each transmit/receive queue of a NIC or virtual port. An additional ring pair connects to the host stack.

After binding a file descriptor to a port, a **netmap** client can send or receive packets in batches through the rings, and possibly implement zero-copy forwarding between ports.

All NICs operating in **netmap** mode use the same memory region, accessible to all processes who own */dev/netmap* file descriptors bound to NICs. Independent **VALE** and **netmap pipe** ports by default use separate memory regions, but can be independently configured to share memory.

ENTERING AND EXITING NETMAP MODE

The following section describes the system calls to create and control **netmap** ports (including **VALE** and **netmap pipe** ports). Simpler, higher level functions are described in the *LIBRARIES* section.

Ports and rings are created and controlled through a file descriptor, created by opening a special device

```
fd = open("/dev/netmap");
```

and then bound to a specific port with an

```
ioctl(fd, NIOCREGIF, (struct nmreq *)arg);
```

netmap has multiple modes of operation controlled by the *struct nmreq* argument. *arg.nr_name* specifies the netmap port name, as follows:

OS network interface name (e.g., ’em0’, ’eth1’, ...)
: the data path of the NIC is disconnected from the host stack, and the file descriptor is bound to the NIC (one or all queues), or to the host stack;

valeSSS:PPP
: the file descriptor is bound to port PPP of VALE switch SSS. Switch instances and ports are dynamically created if necessary.

Both SSS and PPP have the form [0-9a-zA-Z_]+ , the string cannot exceed IFNAMSIZ characters, and PPP cannot be the name of any existing OS network interface.

On return, *arg* indicates the size of the shared memory region, and the number, size and location of all the **netmap** data structures, which can be accessed by mmapping the memory

```
char *mem = mmap(0, arg.nr_memsize, fd);
```

Non-blocking I/O is done with special ioctl(2) select(2) and poll(2) on the file descriptor permit blocking I/O.

While a NIC is in **netmap** mode, the OS will still believe the interface is up and running. OS-generated packets for that NIC end up into a **netmap** ring, and another ring is used to send packets into the OS network stack. A close(2) on the file descriptor removes the binding, and returns the NIC to normal mode (reconnecting the data path to the host stack), or destroys the virtual port.

DATA STRUCTURES

The data structures in the mmapped memory region are detailed in <*sys/net/netmap.h*>, which is the ultimate reference for the **netmap** API. The main structures and fields are indicated below:

struct netmap_if (one per interface)

```
struct netmap_if {
    ...
    const uint32_t   ni_flags;      /* properties              */
    ...
    const uint32_t   ni_tx_rings;   /* NIC tx rings            */
    const uint32_t   ni_rx_rings;   /* NIC rx rings            */
    uint32_t         ni_bufs_head;  /* head of extra bufs list */
    ...
};
```

Indicates the number of available rings (*struct netmap_rings*) and their position in the mmapped region. The number of tx and rx rings (*ni_tx_rings*, *ni_rx_rings*) normally depends on the hardware. NICs also have an extra tx/rx ring pair connected to the host stack. *NIOCREGIF* can also request additional unbound buffers in the same memory space, to be used as temporary storage for packets. The number of extra buffers is specified in the *arg.nr_arg3* field. On success, the kernel writes back to *arg.nr_arg3* the number of extra buffers actually allocated (they may be less than the amount requested if the memory space ran out of buffers). *ni_bufs_head* contains the index of the first of these extra buffers, which are connected in a list (the first uint32_t of each buffer being the index of the next buffer in the list). A 0 indicates the end of the list. The application is free to modify this list and use the buffers (i.e., binding them to the slots of a netmap ring). When closing the netmap file descriptor, the kernel frees the buffers contained in the list pointed by *ni_bufs_head* , irrespectively of the buffers originally provided by the kernel on *NIOCREGIF*.

struct netmap_ring (one per ring)

```
struct netmap_ring {
    ...
    const uint32_t num_slots;   /* slots in each ring            */
    const uint32_t nr_buf_size; /* size of each buffer           */
    ...
    uint32_t       head;        /* (u) first buf owned by user   */
    uint32_t       cur;         /* (u) wakeup position           */
    const uint32_t tail;        /* (k) first buf owned by kernel */
    ...
    uint32_t       flags;
    struct timeval ts;          /* (k) time of last rxsync()     */
    ...
    struct netmap_slot slot[0]; /* array of slots                */
}
```

Implements transmit and receive rings, with read/write pointers, metadata and an array of *slots* describing the buffers.

struct netmap_slot (one per buffer)

```
struct netmap_slot {
    uint32_t buf_idx;           /* buffer index                 */
    uint16_t len;               /* packet length                */
    uint16_t flags;             /* buf changed, etc.            */
    uint64_t ptr;               /* address for indirect buffers */
};
```

Describes a packet buffer, which normally is identified by an index and resides in the mmapped region.

packet buffers
Fixed size (normally 2 KB) packet buffers allocated by the kernel.

The offset of the *struct netmap_if* in the mmapped region is indicated by the *nr_offset* field in the structure returned by NIOCREGIF. From there, all other objects are reachable through relative references (offsets or indexes). Macros and functions in <*net/netmap_user.h*> help converting them into actual pointers:

```
struct netmap_if *nifp = NETMAP_IF(mem, arg.nr_offset);
struct netmap_ring *txr = NETMAP_TXRING(nifp, ring_index);
struct netmap_ring *rxr = NETMAP_RXRING(nifp, ring_index);

char *buf = NETMAP_BUF(ring, buffer_index);
```

RINGS, BUFFERS AND DATA I/O

Rings are circular queues of packets with three indexes/pointers (*head*, *cur*, *tail*); one slot is always kept empty. The ring size (*num_slots*) should not be assumed to be a power of two.

head is the first slot available to userspace;

cur is the wakeup point: select/poll will unblock when *tail* passes *cur*;

tail is the first slot reserved to the kernel.

Slot indexes *must* only move forward; for convenience, the function

```
nm_ring_next(ring, index)
```

returns the next index modulo the ring size.

head and *cur* are only modified by the user program; *tail* is only modified by the kernel. The kernel only reads/writes the *struct netmap_ring* slots and buffers during the execution of a netmap-related system call. The only exception are slots (and buffers) in the range *tail ... head-1*, that are explicitly assigned to the kernel.

TRANSMIT RINGS

On transmit rings, after a **netmap** system call, slots in the range *head ... tail-1* are available for transmission. User code should fill the slots sequentially and advance *head* and *cur* past slots ready to transmit. *cur* may be moved further ahead if the user code needs more slots before further transmissions (see *SCATTER GATHER I/O*).

At the next NIOCTXSYNC/select()/poll(), slots up to *head-1* are pushed to the port, and *tail* may advance if further slots have become available. Below is an example of the evolution of a TX ring:

```
after the syscall, slots between cur and tail are (a)vailable
```

```
             head=cur   tail
              |         |
              v         v
       TX  [.....aaaaaaaaaaa.............]

      user creates new packets to (T)ransmit
               head=cur tail
                  |     |
                  v     v
       TX  [.....TTTTTaaaaaa.............]

      NIOCTXSYNC/poll()/select() sends packets and reports new slots
               head=cur      tail
                  |          |
                  v          v
       TX  [..........aaaaaaaaaaa........]
```

select() and **poll**() will block if there is no space in the ring, i.e.,
    ring->cur == ring->tail
and return when new slots have become available.

High speed applications may want to amortize the cost of system calls by preparing as many packets as possible before issuing them.

A transmit ring with pending transmissions has
    ring->head != ring->tail + 1 (modulo the ring size).
The function *int nm_tx_pending(ring)* implements this test.

RECEIVE RINGS

On receive rings, after a **netmap** system call, the slots in the range *head... tail-1* contain received packets. User code should process them and advance *head* and *cur* past slots it wants to return to the kernel. *cur* may be moved further ahead if the user code wants to wait for more packets without returning all the previous slots to the kernel.

At the next NIOCRXSYNC/select()/poll(), slots up to *head-1* are returned to the kernel for further receives, and *tail* may advance to report new incoming packets.

Below is an example of the evolution of an RX ring:

```
      after the syscall, there are some (h)eld and some (R)eceived slots
```

```
         head  cur     tail
          |     |       |
          v     v       v
     RX  [..hhhhhhRRRRRRRR..........]

    user advances head and cur, releasing some slots and holding others
              head cur  tail
                | |     |
                v  v     v
     RX  [..*****hhhRRRRRR...........]

    NICRXSYNC/poll()/select() recovers slots and reports new packets
              head cur        tail
                | |           |
                v  v           v
     RX  [.......hhhRRRRRRRRRRRR....]
```

SLOTS AND PACKET BUFFERS

Normally, packets should be stored in the netmap-allocated buffers assigned to slots when ports are bound to a file descriptor. One packet is fully contained in a single buffer.

The following flags affect slot and buffer processing:

NS_BUF_CHANGED
: *must* be used when the *buf_idx* in the slot is changed. This can be used to implement zero-copy forwarding, see *ZERO-COPY FORWARDING*.

NS_REPORT
: reports when this buffer has been transmitted. Normally, **netmap** notifies transmit completions in batches, hence signals can be delayed indefinitely. This flag helps detect when packets have been sent and a file descriptor can be closed.

NS_FORWARD
: When a ring is in ’transparent’ mode, packets marked with this flag by the user application are forwarded to the other endpoint at the next system call, thus restoring (in a selective way) the connection between a NIC and the host stack.

NS_NO_LEARN
: tells the forwarding code that the source MAC address for this packet must not be used in the learning bridge code.

NS_INDIRECT
: indicates that the packet's payload is in a user-supplied buffer whose user virtual address is in the 'ptr' field of the slot. The size can reach 65535 bytes.

  This is only supported on the transmit ring of **VALE** ports, and it helps reducing data copies in the interconnection of virtual machines.

NS_MOREFRAG
: indicates that the packet continues with subsequent buffers; the last buffer in a packet must have the flag clear.

SCATTER GATHER I/O

Packets can span multiple slots if the *NS_MOREFRAG* flag is set in all but the last slot. The maximum length of a chain is 64 buffers. This is normally used with **VALE** ports when connecting virtual machines, as they generate large TSO segments that are not split unless they reach a physical device.

NOTE: The length field always refers to the individual fragment; there is no place with the total length of a packet.

On receive rings the macro *NS_RFRAGS(slot)* indicates the remaining number of slots for this packet, including the current one. Slots with a value greater than 1 also have NS_MOREFRAG set.

IOCTLS

netmap uses two ioctls (NIOCTXSYNC, NIOCRXSYNC) for non-blocking I/O. They take no argument. Two more ioctls (NIOCGINFO, NIOCREGIF) are used to query and configure ports, with the following argument:

```
struct nmreq {
    char      nr_name[IFNAMSIZ]; /* (i) port name            */
    uint32_t  nr_version;        /* (i) API version          */
    uint32_t  nr_offset;         /* (o) nifp offset in mmap region */
    uint32_t  nr_memsize;        /* (o) size of the mmap region    */
    uint32_t  nr_tx_slots;       /* (i/o) slots in tx rings  */
    uint32_t  nr_rx_slots;       /* (i/o) slots in rx rings  */
    uint16_t  nr_tx_rings;       /* (i/o) number of tx rings */
    uint16_t  nr_rx_rings;       /* (i/o) number of rx rings */
    uint16_t  nr_ringid;         /* (i/o) ring(s) we care about */
    uint16_t  nr_cmd;            /* (i) special command      */
    uint16_t  nr_arg1;           /* (i/o) extra arguments    */
    uint16_t  nr_arg2;           /* (i/o) extra arguments    */
    uint32_t  nr_arg3;          /* (i/o) extra arguments        */
    uint32_t  nr_flags          /* (i/o) open mode              */
    ...
};
```

A file descriptor obtained through */dev/netmap* also supports the ioctl supported by network devices, see netintro(4).

NIOCGINFO

returns EINVAL if the named port does not support netmap. Otherwise, it returns 0 and (advisory) information about the port. Note that all the information below can change before the interface is actually put in netmap mode.

nr_memsize

indicates the size of the **netmap** memory region. NICs in **netmap** mode all share the same memory region, whereas **VALE** ports have independent regions for each port.

nr_tx_slots, *nr_rx_slots*

indicate the size of transmit and receive rings.

nr_tx_rings, *nr_rx_rings*

indicate the number of transmit and receive rings. Both ring number and sizes may be configured at runtime using interface-specific functions (e.g., ethtool(8)).

NIOCREGIF

binds the port named in *nr_name* to the file descriptor. For a physical device this also switches it into **netmap** mode, disconnecting it from the host stack. Multiple file descriptors can be bound to the same port, with proper synchronization left to the user.

The recommended way to bind a file descriptor to a port is to use function *nm_open(..)* (see *LIBRARIES*) which parses names to access specific port types and enable features. In the following we document the main features.

NIOCREGIF can also bind a file descriptor to one endpoint of a *netmap pipe*, consisting of two netmap ports with a crossover connection. A netmap pipe share the same memory space of the parent port, and is meant to enable configuration where a master process acts as a dispatcher towards slave processes.

To enable this function, the *nr_arg1* field of the structure can be used as a hint to the kernel to indicate how many pipes we expect to use, and reserve extra space in the memory region.

On return, it gives the same info as NIOCGINFO, with *nr_ringid* and *nr_flags* indicating the identity of the rings controlled through the file descriptor.

nr_flags nr_ringid selects which rings are controlled through this file descriptor. Possible values of *nr_flags* are indicated below, together with the naming schemes that application libraries (such as the **nm_open** indicated below) can use to indicate the specific set of rings. In the example below, "netmap:foo" is any valid netmap port name.

NR_REG_ALL_NIC netmap:foo
: (default) all hardware ring pairs

NR_REG_SW netmap:foo^
: the ‘‘host rings’’, connecting to the host stack.

NR_REG_NIC_SW netmap:foo*
: all hardware rings and the host rings

NR_REG_ONE_NIC netmap:foo-i
: only the i-th hardware ring pair, where the number is in *nr_ringid*;

NR_REG_PIPE_MASTER netmap:foo{i
: the master side of the netmap pipe whose identifier (i) is in *nr_ringid*;

NR_REG_PIPE_SLAVE netmap:foo}i
: the slave side of the netmap pipe whose identifier (i) is in *nr_ringid*.

: The identifier of a pipe must be thought as part of the pipe name, and does not need to be sequential. On return the pipe will only have a single ring pair with index 0, irrespective of the value of *i*.

By default, a poll(2) or select(2) call pushes out any pending packets on the transmit ring, even if no write events are specified. The feature can be disabled by or-ing *NETMAP_NO_TX_POLL* to the value written to *nr_ringid*. When this feature is used, packets are transmitted only on *ioctl(NIOCTXSYNC)* or *select() / poll()* are called with a write event (POLLOUT/wfdset) or a full ring.

When registering a virtual interface that is dynamically created to a **VALE** switch, we can specify the desired number of rings (1 by default, and currently up to 16) on it using nr_tx_rings and nr_rx_rings fields.

NIOCTXSYNC
: tells the hardware of new packets to transmit, and updates the number of slots available for transmission.

NIOCRXSYNC
: tells the hardware of consumed packets, and asks for newly available packets.

SELECT, POLL, EPOLL, KQUEUE

select(2) and poll(2) on a **netmap** file descriptor process rings as indicated in *TRANSMIT RINGS* and *RECEIVE RINGS*, respectively when write (POLLOUT) and read (POLLIN) events are requested. Both block if no slots are available in the ring (*ring->cur == ring->tail*). Depending on the platform, epoll(7) and kqueue(2) are supported too.

Packets in transmit rings are normally pushed out (and buffers reclaimed) even without requesting write events. Passing the NETMAP_NO_TX_POLL flag to *NIOCREGIF* disables this feature. By default, receive rings are processed only if read events are requested. Passing the NETMAP_DO_RX_POLL flag to *NIOCREGIF updates receive rings even without read events.* Note that on epoll(7) and kqueue(2), NETMAP_NO_TX_POLL and NETMAP_DO_RX_POLL only have an effect when some event is posted for the file descriptor.

LIBRARIES

The **netmap** API is supposed to be used directly, both because of its simplicity and for efficient integration with applications.

For convenience, the <*net/netmap_user.h*> header provides a few macros and functions to ease creating a file descriptor and doing I/O with a **netmap** port. These are loosely modeled after the pcap(3) API, to ease porting of libpcap-based applications to **netmap**. To use these extra functions, programs should

```
#define NETMAP_WITH_LIBS
```

before

```
#include <net/netmap_user.h>
```

The following functions are available:

*struct nm_desc * nm_open(const char *ifname, const struct nmreq *req, uint64_t flags, const struct nm_desc *arg*)
: similar to pcap_open_live(3), binds a file descriptor to a port.

  ifname
  : is a port name, in the form "netmap:PPP" for a NIC and "valeSSS:PPP" for a **VALE** port.

req provides the initial values for the argument to the NIOCREGIF ioctl. The nm_flags and nm_ringid values are overwritten by parsing ifname and flags, and other fields can be overridden through the other two arguments.

arg points to a struct nm_desc containing arguments (e.g., from a previously open file descriptor) that should override the defaults. The fields are used as described below

flags
can be set to a combination of the following flags: *NETMAP_NO_TX_POLL*, *NETMAP_DO_RX_POLL* (copied into nr_ringid); *NM_OPEN_NO_MMAP* (if arg points to the same memory region, avoids the mmap and uses the values from it); *NM_OPEN_IFNAME* (ignores ifname and uses the values in arg); *NM_OPEN_ARG1*, *NM_OPEN_ARG2*, *NM_OPEN_ARG3* (uses the fields from arg); *NM_OPEN_RING_CFG* (uses the ring number and sizes from arg).

*int nm_close(struct nm_desc *d*)
closes the file descriptor, unmaps memory, frees resources.

*int nm_inject(struct nm_desc *d, const void *buf, size_t size*)
similar to *pcap_inject()*, pushes a packet to a ring, returns the size of the packet is successful, or 0 on error;

*int nm_dispatch(struct nm_desc *d, int cnt, nm_cb_t cb, u_char *arg*)
similar to *pcap_dispatch()*, applies a callback to incoming packets

*u_char * nm_nextpkt(struct nm_desc *d, struct nm_pkthdr *hdr*)
similar to *pcap_next()*, fetches the next packet

SUPPORTED DEVICES

netmap natively supports the following devices:

On FreeBSD: cxgbe(4), em(4), iflib(4) (providing igb(4) and em(4)), ixgbe(4), ixl(4), re(4), vtnet(4).

On Linux e1000, e1000e, i40e, igb, ixgbe, ixgbevf, r8169, virtio_net, vmxnet3.

NICs without native support can still be used in **netmap** mode through emulation. Performance is inferior to native netmap mode but still significantly higher than various raw socket types (bpf, PF_PACKET, etc.). Note that for slow devices (such as 1 Gbit/s and slower NICs, or several 10 Gbit/s NICs whose hardware is unable to sustain line rate), emulated and native mode will likely have similar or same throughput.

When emulation is in use, packet sniffer programs such as tcpdump could see received packets before they are diverted by netmap. This behaviour is not intentional, being just an artifact of the implementation of emulation. Note that in case the netmap application subsequently moves packets received from the emulated adapter onto the host RX ring, the sniffer will intercept those packets again, since the packets are injected to the host stack as they were received by the network interface.

Emulation is also available for devices with native netmap support, which can be used for testing or performance comparison. The sysctl variable *dev.netmap.admode* globally controls how netmap mode is implemented.

SYSCTL VARIABLES AND MODULE PARAMETERS

Some aspects of the operation of **netmap** and **VALE** are controlled through sysctl variables on FreeBSD (*dev.netmap.**) and module parameters on Linux (*/sys/module/netmap/parameters/**):

dev.netmap.admode: 0

Controls the use of native or emulated adapter mode.

0 uses the best available option;

1 forces native mode and fails if not available;

2 forces emulated hence never fails.

dev.netmap.generic_rings: 1

Number of rings used for emulated netmap mode

dev.netmap.generic_ringsize: 1024

Ring size used for emulated netmap mode

dev.netmap.generic_mit: 100000

Controls interrupt moderation for emulated mode

dev.netmap.fwd: 0

Forces NS_FORWARD mode

dev.netmap.txsync_retry: 2

Number of txsync loops in the **VALE** flush function

dev.netmap.no_pendintr: 1

Forces recovery of transmit buffers on system calls

dev.netmap.no_timestamp: 0
Disables the update of the timestamp in the netmap ring

dev.netmap.verbose: 0
Verbose kernel messages

dev.netmap.buf_num: 163840

dev.netmap.buf_size: 2048

dev.netmap.ring_num: 200

dev.netmap.ring_size: 36864

dev.netmap.if_num: 100

dev.netmap.if_size: 1024
Sizes and number of objects (netmap_if, netmap_ring, buffers) for the global memory region. The only parameter worth modifying is *dev.netmap.buf_num* as it impacts the total amount of memory used by netmap.

dev.netmap.buf_curr_num: 0

dev.netmap.buf_curr_size: 0

dev.netmap.ring_curr_num: 0

dev.netmap.ring_curr_size: 0

dev.netmap.if_curr_num: 0

dev.netmap.if_curr_size: 0
Actual values in use.

dev.netmap.priv_buf_num: 4098

dev.netmap.priv_buf_size: 2048

dev.netmap.priv_ring_num: 4

dev.netmap.priv_ring_size: 20480

dev.netmap.priv_if_num: 2

dev.netmap.priv_if_size: 1024
Sizes and number of objects (netmap_if, netmap_ring, buffers) for private memory regions. A separate memory region is used for each **VALE** port and each pair of **netmap pipes**.

dev.netmap.bridge_batch: 1024
Batch size used when moving packets across a **VALE** switch. Values above 64 generally guarantee good performance.

dev.netmap.max_bridges: 8
Max number of **VALE** switches that can be created. This tunable can be specified at loader time.

dev.netmap.ptnet_vnet_hdr: 1
Allow ptnet devices to use virtio-net headers

SYSTEM CALLS

netmap uses select(2), poll(2), epoll(7) and kqueue(2) to wake up processes when significant events occur, and mmap(2) to map memory. ioctl(2) is used to configure ports and **VALE switches**.

Applications may need to create threads and bind them to specific cores to improve performance, using standard OS primitives, see pthread(3). In particular, pthread_setaffinity_np(3) may be of use.

EXAMPLES

TEST PROGRAMS

netmap comes with a few programs that can be used for testing or simple applications. See the *examples/* directory in **netmap** distributions, or *tools/tools/netmap/* directory in FreeBSD distributions.

pkt-gen(8) is a general purpose traffic source/sink.

As an example
```
pkt-gen -i ix0 -f tx -l 60
```
can generate an infinite stream of minimum size packets, and
```
pkt-gen -i ix0 -f rx
```
is a traffic sink. Both print traffic statistics, to help monitor how the system performs.

pkt-gen(8) has many options can be uses to set packet sizes, addresses, rates, and use multiple send/receive threads and cores.

bridge(4) is another test program which interconnects two **netmap** ports. It can be used for transparent forwarding between interfaces, as in

```
bridge -i netmap:ix0 -i netmap:ix1
```

or even connect the NIC to the host stack using netmap

```
bridge -i netmap:ix0
```

USING THE NATIVE API

The following code implements a traffic generator:

```
#include <net/netmap_user.h>
...
void sender(void)
{
    struct netmap_if *nifp;
    struct netmap_ring *ring;
    struct nmreq nmr;
    struct pollfd fds;

    fd = open("/dev/netmap", O_RDWR);
    bzero(&nmr, sizeof(nmr));
    strcpy(nmr.nr_name, "ix0");
    nmr.nm_version = NETMAP_API;
    ioctl(fd, NIOCREGIF, &nmr);
    p = mmap(0, nmr.nr_memsize, fd);
    nifp = NETMAP_IF(p, nmr.nr_offset);
    ring = NETMAP_TXRING(nifp, 0);
    fds.fd = fd;
    fds.events = POLLOUT;
    for (;;) {
            poll(&fds, 1, -1);
            while (!nm_ring_empty(ring)) {
                i = ring->cur;
                buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
                ... prepare packet in buf ...
                ring->slot[i].len = ... packet length ...
                ring->head = ring->cur = nm_ring_next(ring, i);
            }
    }
}
```

HELPER FUNCTIONS

A simple receiver can be implemented using the helper functions:

```
#define NETMAP_WITH_LIBS
#include <net/netmap_user.h>
...
void receiver(void)
{
    struct nm_desc *d;
    struct pollfd fds;
    u_char *buf;
    struct nm_pkthdr h;
    ...
    d = nm_open("netmap:ix0", NULL, 0, 0);
    fds.fd = NETMAP_FD(d);
    fds.events = POLLIN;
    for (;;) {
            poll(&fds, 1, -1);
        while ( (buf = nm_nextpkt(d, &h)) )
                consume_pkt(buf, h.len);
    }
    nm_close(d);
}
```

ZERO-COPY FORWARDING

Since physical interfaces share the same memory region, it is possible to do packet forwarding between ports swapping buffers. The buffer from the transmit ring is used to replenish the receive ring:

```
    uint32_t tmp;
    struct netmap_slot *src, *dst;
    ...
    src = &src_ring->slot[rxr->cur];
    dst = &dst_ring->slot[txr->cur];
    tmp = dst->buf_idx;
    dst->buf_idx = src->buf_idx;
    dst->len = src->len;
    dst->flags = NS_BUF_CHANGED;
    src->buf_idx = tmp;
    src->flags = NS_BUF_CHANGED;
    rxr->head = rxr->cur = nm_ring_next(rxr, rxr->cur);
```

```
txr->head = txr->cur = nm_ring_next(txr, txr->cur);
...
```

ACCESSING THE HOST STACK

The host stack is for all practical purposes just a regular ring pair, which you can access with the netmap API (e.g., with

```
nm_open("netmap:eth0^", ...);
```

All packets that the host would send to an interface in **netmap** mode end up into the RX ring, whereas all packets queued to the TX ring are send up to the host stack.

VALE SWITCH

A simple way to test the performance of a **VALE** switch is to attach a sender and a receiver to it, e.g., running the following in two different terminals:

```
pkt-gen -i vale1:a -f rx # receiver
pkt-gen -i vale1:b -f tx # sender
```

The same example can be used to test netmap pipes, by simply changing port names, e.g.,

```
pkt-gen -i vale2:x{3 -f rx # receiver on the master side
pkt-gen -i vale2:x}3 -f tx # sender on the slave side
```

The following command attaches an interface and the host stack to a switch:

```
valectl -h vale2:em0
```

Other **netmap** clients attached to the same switch can now communicate with the network card or the host.

SEE ALSO

vale(4), bridge(8), valectl(8), lb(8), nmreplay(8), pkt-gen(8)

http://info.iet.unipi.it/~luigi/netmap/

Luigi Rizzo, Revisiting network I/O APIs: the netmap framework, Communications of the ACM, 55 (3), pp.45-51, March 2012

Luigi Rizzo, netmap: a novel framework for fast packet I/O, Usenix ATC'12, June 2012, Boston

Luigi Rizzo, Giuseppe Lettieri, VALE, a switched ethernet for virtual machines, ACM CoNEXT'12, December 2012, Nice

Luigi Rizzo, Giuseppe Lettieri, Vincenzo Maffione, Speeding up packet I/O in virtual machines, ACM/IEEE ANCS'13, October 2013, San Jose

AUTHORS

The **netmap** framework has been originally designed and implemented at the Universita‘ di Pisa in 2011 by Luigi Rizzo, and further extended with help from Matteo Landi, Gaetano Catalli, Giuseppe Lettieri, and Vincenzo Maffione.

netmap and **VALE** have been funded by the European Commission within FP7 Projects CHANGE (257422) and OPENLAB (287581).

CAVEATS

No matter how fast the CPU and OS are, achieving line rate on 10G and faster interfaces requires hardware with sufficient performance. Several NICs are unable to sustain line rate with small packet sizes. Insufficient PCIe or memory bandwidth can also cause reduced performance.

Another frequent reason for low performance is the use of flow control on the link: a slow receiver can limit the transmit speed. Be sure to disable flow control when running high speed experiments.

SPECIAL NIC FEATURES

netmap is orthogonal to some NIC features such as multiqueue, schedulers, packet filters.

Multiple transmit and receive rings are supported natively and can be configured with ordinary OS tools, such as ethtool(8) or device-specific sysctl variables. The same goes for Receive Packet Steering (RPS) and filtering of incoming traffic.

netmap *does not use* features such as *checksum offloading*, *TCP segmentation offloading*, *encryption*, *VLAN encapsulation/decapsulation*, etc. When using netmap to exchange packets with the host stack, make sure to disable these features.