An introduction to Netmap

Giuseppe Lettieri

Dipartimento di Ingegneria dell’Informazione
Università di Pisa

SIGCOMM’17, Los Angeles, CA, 27 August 2017
What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iel.unipi.it/ luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap
What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iet.unipi.it/luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap
What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iit.unipi.it/ luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap
What is netmap?

- A framework for high speed packet I/O
- Initially designed and developed by Luigi Rizzo at University of Pisa (http://info.iet.unipi.it/luigi/netmap/)
- Part of FreeBSD, external module for Linux and Windows
- Code available on github: https://github.com/luigirizzo/netmap
netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC’s hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines
netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC’s hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines
netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC’s hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines
netmap numbers

- line rate for 10G with minimum sized packets using a fraction of a core
- over 30 Mpps on 40G NICs (limited by the NIC’s hardware)
- over 20 Mpps on VALE ports (software switch)
- over 100 Mpps on netmap pipes
- almost the same on bare metal or virtual machines
line rate for 10G with minimum sized packets using a fraction of a core
over 30 Mpps on 40G NICs (limited by the NIC’s hardware)
over 20 Mpps on VALE ports (software switch)
over 100 Mpps on netmap pipes
almost the same on bare metal or virtual machines
let us consider the RX path
at open time, the driver fills the RX ring with empty skbufts
the ring slots that the NIC can use are in the \([\text{RDH}, \text{RDT})\) interval
Background: RX in traditional OS

when a new message is received
the NIC copies it into the first available skbuf
it updates the head pointer and possibly sends an interrupt to notify the driver
the driver eventually notices the new message and moves the skbuf up the stack
the driver allocates a new empty skbuf to fill the ring again
it then moves the tail to make the new skbuf available to the NIC
the message is eventually copied to userspace and the containing skbuf is discarded
many messages may be received before the driver starts processing the queue
many messages may be received before the driver starts processing the queue
many messages may be received before the driver starts processing the queue
many messages may be received before the driver starts processing the queue
Background: RX in traditional OS

many messages may be received before the driver starts processing the queue
many messages may be received before the driver starts processing the queue
Background: RX in traditional OS

A single update of the head pointer will reveal all the new messages.
the driver will process all of them, possibly in a single run
the driver will process all of them, possibly in a single run
the driver will process all of them, possibly in a single run
the tail pointer can be updated a single time
the user will still have to copy each one of the messages, possibly via several system calls
the user will still have to copy each one of the messages, possibly via several system calls
the user will still have to copy each one of the messages, possibly via several system calls
let us now consider the TX path
at open time, the TX ring is empty
this is signaled by $\text{TDT} = \text{TDH}$
Background: TX in traditional OS

assume an application sends a new message through a socket
the kernel allocates an skbuf where it copies the message, then pushes it down the stack until it reaches the driver

network stack

tdt tdh
the driver links the skbuf in the TX ring
then it notifies the NIC by updating the ring tail
the NIC reads the message via DMA and sends it over the link
when the DMA is completed the NIC updates the ring head and possibly sends an interrupt
the driver eventually notices and frees the skb
the driver may push several skb’s in the queue
the driver may push several skb’s in the queue
the driver may push several skb’s in the queue
the driver may push several skb’s in the queue
a single update of TDT may notify several messages
other messages may be enqueued while the driver is sending the previous ones
other messages may be enqueued while the driver is sending the previous ones
other messages may be enqueued while the driver is sending the previous ones
other messages may be enqueued while the driver is sending the previous ones
all completed skb’s may be freed in a single run of the driver
let us now consider a NIC open in netmap mode
when we open a NIC in netmap mode, the NIC is disconnected from the host stack
netmap buffers and rings are allocated in shared memory.
Netmap mode

The netmap rings are pre-filled with netmap buffers.

Network Stack

Netmap ring

Netmap buffers

Netmap ring

Netmap buffers
and the NIC rings are made to share the same buffers
Netmap mode

the NIC only has access to its rings and the netmap buffers
the netmap application has access to the netmap rings and buffers
the netmap rings also have *head* and *tail* pointers. Netmap applications may access the slots and buffers in [head, tail)
For the RX ring, this interval contains *new packets*. Initially, it is empty...
Netmap mode

...and all buffers belong to the NIC
Assume a packet is received by the NIC
Netmap mode

The NIC copies it into the first available netmap buffer and notifies the netmap application.
When the netmap application orders a *ring sync*, the tail pointer is updated to reflect the new state.
Now the netmap application may read the new message. When it is done, it moves head.
The next time that it orders a ring sync, the NIC tail pointer is updated.
Netmap mode

Batching is possible: assume two new packets arrive
The netmap application orders a sync and tail now reveals both packets.
While the application processes the new packets, other may arrive
While the application processes the new packets, other may arrive
Netmap mode

When the application orders a sync again, both tail and rdt are updated.
For the TX ring, the \([\text{head}, \text{tail}]\) interval contains *empty buffers*. Initially, it is full.
Netmap mode

The NIC has initially no buffers

network stack

G. Lettieri  Netmap intro
The netmap application may prepare several packets in the empty buffers...
The netmap application may prepare several packets in the empty buffers...
Netmap mode

The netmap application may prepare several packets in the empty buffers...
The netmap application may prepare several packets in the empty buffers...
... and then orders a sync. The kernel will update tdt, therefore notifying the NIC, which will start transmitting
...and then orders a sync. The kernel will update tdt, therefore notifying the NIC, which will start transmitting
Netmap mode

meanwhile, the application may prepare new packets
Assume two packets have been transmitted when the app orders a sync again.
Now, both tail and tdt are updated
The netmap user data-structures

Defined (and documented!) in /usr/include/net/netmap.h.
What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in `[head, tail]`
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let `cur = head`. 
What is \textit{cur}?  

- Another user-controlled pointer in the ring  
- It must lie in \([\text{head},\text{tail}]\)  
- It moves past the slots that the application has seen, without returning them to netmap  
- In RX rings:  
  - hold packets that you have seen but not yet finished to process  
- in TX rings:  
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)  
- in most cases, just let \(\text{cur} = \text{head}\).
What is $\text{cur}$?

- Another user-controlled pointer in the ring
- It must lie in $[\text{head}, \text{tail}]$
- It moves past the slots that the application has seen, without returning them to netmap

- In RX rings:
  - hold packets that you have seen but not yet finished to process
- in TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)

- in most cases, just let $\text{cur} = \text{head}$. 

What is `cur`?

- Another user-controlled pointer in the ring
- It must lie in `[head, tail]`
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- In TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- In most cases, just let `cur = head`. 
What is \textit{cur}?

- Another user-controlled pointer in the ring
- It must lie in \([\text{head}, \text{tail}]\)
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- In TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- In most cases, just let \textit{cur} = \textit{head}.
What is \texttt{cur}?

- Another user-controlled pointer in the ring
- It must lie in \([\text{head}, \text{tail}]\)
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- In TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let \(\texttt{cur} = \text{head}\).
What is \textit{cur}?

- Another user-controlled pointer in the ring
- It must lie in \([\text{head, tail}]\)
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- In TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- in most cases, just let \(\text{cur} = \text{head}\).
What is cur?

- Another user-controlled pointer in the ring
- It must lie in \([\text{head}, \text{tail}]\)
- It moves past the slots that the application has seen, without returning them to netmap
- In RX rings:
  - hold packets that you have seen but not yet finished to process
- In TX rings:
  - the available slots are not sufficient and you need to wait for more (e.g., you have to send a multi-slot packet)
- In most cases, just let \(\text{cur} = \text{head}\).
The simplified setup API

Include libs

```c
#include <net/netmap.h>
#define NETMAP_WITH_LIBS
#include <net/netmap_user.h>
```

Open a port in netmap mode

```c
struct nm_des *nmd = nm_open("netmap:eth0", NULL, 0, NULL);
```

Reach your netmap_if

```c
struct netmap_if *nifp = nmd->nifp;
```
The simplified setup API

Include libs

```c
#include <net/netmap.h>
#define NETMAP_WITH_LIBS
#include <net/netmap_user.h>
```

Open a port in netmap mode

```c
struct nm_des *nmd = 
    nm_open("netmap:eth0", NULL, 0, NULL);
```

Reach your netmap_if

```c
struct netmap_if *nifp = nmd->nifp;
```
The simplified setup API

Include libs

```c
#include <net/netmap.h>
#define NETMAP_WITH_LIBS
#include <net/netmap_user.h>
```

Open a port in netmap mode

```c
struct nm_des *nmd = 
m_open("netmap:eth0", NULL, 0, NULL);
```

Reach your netmap_if

```c
struct netmap_if *nifp = nmd->nifp;
```
A first look at \texttt{nm\_open()}

```c
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if \texttt{ifname} is “\texttt{netmap:if}” it
  - puts \texttt{if} in netmap mode, if necessary
  - \texttt{mmap()}s the netmap user structures into the process address space
  - returns an \texttt{nm\_desc} with all the info
- on error it returns NULL and sets \texttt{errno} (\texttt{errno = 0} means \texttt{ifname} not recognized)
A first look at `nm_open()`

```
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if `ifname` is “netmap:if” it
  1. puts `if` in netmap mode, if necessary
  2. `mmap()`s the netmap user structures into the process address space
  3. returns an `nm_desc` with all the info
- on error it returns NULL and sets `errno` (`errno = 0` means `ifname` not recognized)
A first look at `nm_open()`

```c
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if `ifname` is “netmap:if” it
  1. puts `if` in netmap mode, if necessary
  2. `mmap()`s the netmap user structures into the process address space
  3. returns an `nm_desc` with all the info
- on error it returns NULL and sets `errno` (`errno = 0` means `ifname` not recognized)
struct nm_desc *
nm_open(const char *ifname, /* other args */);

- use NULL, 0 and NULL for other args
- if ifname is “netmap:if” it
  1. puts if in netmap mode, if necessary
  2. mmap()s the netmap user structures into the process address space
  3. returns an nm_desc with all the info
- on error it returns NULL and sets errno (errno = 0 means ifname not recognized)
A first look at \texttt{nm\_open()}


define \texttt{nm\_desc} *
\texttt{nm\_open(const char *ifname, /* other args */);}

- use \texttt{NULL}, \texttt{0} and \texttt{NULL} for other args
- if \texttt{ifname} is “netmap:\texttt{if}” it
  1. \texttt{puts if} in netmap mode, if necessary
  2. \texttt{mmap()}s the netmap user structures into the process address space
  3. returns an \texttt{nm\_desc} with all the info
- on error it returns \texttt{NULL} and sets \texttt{errno} (\texttt{errno = 0} means \texttt{ifname} not recognized)
A first look at `nm_open()`

```c
struct nm_desc *
nm_open(const char *ifname, /* other args */);
```

- use NULL, 0 and NULL for other args
- if `ifname` is “netmap:if” it
  1. puts `if` in netmap mode, if necessary
  2. `mmap()`s the netmap user structures into the process address space
  3. returns an `nm_desc` with all the info
- on error it returns NULL and sets `errno` (`errno = 0` means `ifname` not recognized)
Working with the rings

Once you have the pointer to `netmap_if` you can reach the rings:

```c
struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
```

There may be many rings! Use the `{first,last}_{tx,rx}_ring fields in the nmd returned by `nm_open()`

- there are `ring->num_slots` slots in the `ring->slots[]` array
- to obtain the buffer pointed to by a slot of `ring`:
  ```c
  void *buf = NETMAP_BUF(ring, slot->buf_idx);
  ```
Working with the rings

Once you have the pointer to `netmap_if` you can reach the rings:

```c
struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
```

There may be many rings! Use the `{first, last}_{tx, rx}_ring fields in the nmd returned by `nm_open()`

- there are `ring->num_slots` slots in the `ring->slots[]` array
- to obtain the buffer pointed to by a slot of `ring`:
  ```c
  void *buf = NETMAP_BUF(ring, slot->buf_idx);
  ```
Once you have the pointer to `netmap_if` you can reach the rings:

```c
struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
```

There may be many rings! Use the `{first, last}`_{tx, rx}_ring fields in the nmd returned by `nm_open()`

- there are `ring->num_slots` slots in the `ring->slots[]` array
- to obtain the buffer pointed to by a slot of `ring`:

```c
void *buf = NETMAP_BUF(ring, slot->buf_idx);
```
Working with the rings

- Once you have the pointer to `netmap_if` you can reach the rings:
  ```c
  struct netmap_ring *rxring = NETMAP_RXRING(nifp, 0);
  struct netmap_ring *rtring = NETMAP_TXRING(nifp, 0);
  ```

- There may be many rings! Use the `{first, last}` `{tx, rx}` ring fields in the `nmd` returned by `nm_open()`

- There are `ring->num_slots` slots in the `ring->slots[]` array

- To obtain the buffer pointed to by a slot of `ring`:
  ```c
  void *buf = NETMAP_BUF(ring, slot->buf_idx);
  ```
Sync-ing the rings: non-blocking

RX rings

```
ioctl(nmd->fd, NIOCRXSYNC);
```

Sync all registered RX rings;

TX rings

```
ioctl(nmd->fd, NIOCTXSYNC);
```

Sync all registered TX rings;

Can be used to implement busy-polling.
Sync-ing the rings: non-blocking

<table>
<thead>
<tr>
<th>RX rings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ioctl(nmd-&gt;fd, NIOCRXSYNC);</code></td>
</tr>
<tr>
<td>Sync all registered RX rings;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TX rings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ioctl(nmd-&gt;fd, NIOCTXSYNC);</code></td>
</tr>
<tr>
<td>Sync all registered TX rings;</td>
</tr>
</tbody>
</table>

Can be used to implement busy-polling.
Sync-ing the rings: non-blocking

RX rings

```c
ioctl(nmd->fd, NIOCRXSYNC);
```

Sync all registered RX rings;

TX rings

```c
ioctl(nmd->fd, NIOCTXSYNC);
```

Sync all registered TX rings;

Can be used to implement busy-polling.
Other useful functions

- **nm_ring_next:** (struct netmap_ring *, unint32_t): increment ring pointers with wrap around
- **nm_ring_empty:** (struct netmap_ring *): true iff there are no new packets (RX ring) or slots available for sending (TX ring)
- **nm_ring_space:** (struct netmap_ring *): number of slots available for sending (TX ring) or number of received packets (RX ring)
- **nm_close:** (struct nm_desc *): pass the nmd returned by nm_open() to clean up and free it
First example: busy-polling sink

Write a netmap application that
- accepts two arguments from the command line:
  - the name of a netmap port
  - an UDP port number
- it opens the netmap port and counts the received UDP packets with the given port number

Boilerplate code already available in
codelab/sink.c
Sync-ing the rings: blocking

**using poll()**

```c
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if POLLIN
- sync on the registered TX rings if POLLOUT OR there are packets to send
- if all rings are empty (cur = tail), block
- Note: no need to sync again when poll() returns!
Sync-ing the rings: blocking

**using** `poll()`

```c
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if POLLIN
- sync on the registered TX rings if POLLOUT *OR* there are packets to send
- if all rings are empty (cur = tail), block
- Note: no need to sync again when `poll()` returns!
Sync-ing the rings: blocking

using `poll()`

```c
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if POLLIN
- sync on the registered TX rings if POLLOUT OR there are packets to send
- if all rings are empty (`cur = tail`), block
- Note: no need to sync again when `poll()` returns!
Sync-ing the rings: blocking

using `poll()`

```c
struct pollfd pfd = {
    .fd = nmd->fd,
    .events = POLLIN /* and/or POLLOUT */
};

int i = poll(pfd, 1, timeout);
```

- sync the registered empty RX rings if POLLIN
- sync on the registered TX rings if POLLOUT OR there are packets to send
- if all rings are empty (cur = tail), block
- Note: no need to sync again when `poll()` returns!
Modify the first example to use `poll()` instead of `ioctl()`.
Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call.
- Don’t `poll()` needlessly, or you fall back to busy wait.
- TX is subtle:
  - You need to `POLLOUT` only if you are out of TX slots.
  - But also to start TX as a side effect.
  - In default mode, TX will start also if you `POLLIN`.
  - (Latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`.)
Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don’t `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

G. Lettieri
Netmap intro
Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don’t `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

G. Lettieri
Netmap intro
Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don’t `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)
You can \texttt{nm\_open()} several ports and \texttt{poll()} all of them in a single call.

Don’t \texttt{poll()} needlessly, or you fall back to busy wait.

\textbf{TX is subtle:}

- you need to \texttt{POLLOUT} only if you are out of TX slots
- but also to start TX as a side effect
- in default mode, TX will start also if you \texttt{POLLIN}
- (latter can be disabled by passing the \texttt{NETMAP\_NO\_TX\_POLL} flag to \texttt{nm\_open()})
Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don’t `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

G. Lettieri  Netmap intro
Polling on several ports and mixed directions

- You can `nm_open()` several ports and `poll()` all of them in a single call
- Don’t `poll()` needlessly, or you fall back to busy wait
- TX is subtle:
  - you need to `POLLOUT` only if you are out of TX slots
  - but also to start TX as a side effect
  - in default mode, TX will start also if you `POLLIN`
  - (latter can be disabled by passing the `NETMAP_NO_TX_POLL` flag to `nm_open()`)

G. Lettieri
Netmap intro
Third example: forward

Write a netmap application that receives two netmap port names and an UDP destination port number from the command line. Then the application

- forwards from the first to the second port all the UDP packets with the given destination port
- drops all other packets
- if the destination port is 0, it forwards all packets

Boilerplate in

codelab/forward.c
Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a buf_idx of a slot, remember to set the NS_BUF_CHANGED flag in the slot flags

Only possible if rings and buffers are in the same netmap memory region
Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a buf_idx of a slot, remember to set the NS_BUF_CHANGED flag in the slot flags

Only possible if rings and buffers are in the same netmap memory region
Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a buf_idx of a slot, remember to set the NS_BUF_CHANGED flag in the slot flags

Only possible if rings and buffers are in the same netmap memory region
Zero-copy

- just swap the buffers between the slots of two rings
- whenever you change a buf_idx of a slot, remember to set the NS_BUF_CHANGED flag in the slot flags

Only possible if rings and buffers are in the same *netmap memory region*
What are netmap memory regions?

- ifs
- rings
- buffers

- System memory pre-allocated by netmap
- May be shared by many ports
- There may be more than one region

**Beware**

You always get access to *entire* regions, not just to the resources allocated by your request.
What are netmap memory regions?

- `ifs`
- `rings` - system memory pre-allocated by netmap
- `buffers` - may be shared by many ports
  - there may be more than one region

**Beware**

you always get access to *entire* regions, not just to the resources allocated by your request.
What are netmap memory regions?

- `ifs`
- `rings`
  - system memory pre-allocated by netmap
  - may be shared by many ports
  - there may be more than one region

**Beware**
you always get access to entire regions, not just to the resources allocated by your request.
What are netmap memory regions?

- **ifs**
- **rings**
  - system memory pre-allocated by netmap
  - may be shared by many ports
  - there may be more than one region
- **buffers**

**Beware**

you always get access to *entire* regions, not just to the resources allocated by your request.
What are netmap memory regions?

- **ifs**
- **rings**
  - system memory pre-allocated by netmap
  - may be shared by many ports
  - there may be more than one region
- **buffers**

**Beware**

You always get access to *entire* regions, not just to the resources allocated by your request.
when a port is put in netmap mode, netmap selects a memory region for the user data-structures

- legacy behaviour: by default all “hardware” ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region
Memory regions and ports

- when a port is put in netmap mode, netmap selects a memory region for the user data-structures
- legacy behaviour: by default all “hardware” ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region
Memory regions and ports

- When a port is put in netmap mode, netmap selects a memory region for the user data-structures.
- Legacy behaviour: by default all "hardware" ports use the same (global) region.
- PROS of sharing: zero-copy
- CONS: no protection!
- The user may select a different region.
Memory regions and ports

- when a port is put in netmap mode, netmap selects a memory region for the user data-structures
- legacy behaviour: by default all “hardware” ports use the same (global) region
- PROS of sharing: zero-copy
- CONS: no protection!
- the user may select a different region
when a port is put in netmap mode, netmap selects a memory region for the user data-structures

legacy behaviour: by default all “hardware” ports use the same (global) region

PROS of sharing: zero-copy

CONS: no protection!

the user may select a different region
nm_open() and memory regions

**first port**

nmd1 = nm_open("netmap:eth0", NULL, 0, NULL);

**second port**

nmd2 = nm_open("netmap:eth1", NULL, NM_OPEN_NO_MMAP, nmd1);

The two ports are in the same region iff

nmd1->mem == nmd2->mem
**nm_open() and memory regions**

**first port**

```c
nmd1 = nm_open("netmap:eth0", NULL, 0, NULL);
```

**second port**

```c
nmd2 = nm_open("netmap:eth1", NULL,
               NM_OPEN_NO_MMAP, nmd1);
```

The two ports are in the same region iff

```c
nmd1->mem == nmd2->mem
```
nm_open() and memory regions

**first port**

```c
nmd1 = nm_open("netmap:eth0", NULL, 0, NULL);
```

**second port**

```c
nmd2 = nm_open("netmap:eth1", NULL,
               NM_OPEN_NO_MMAP, nmd1);
```

The two ports are in the same region iff

```c
nmd1->mem == nmd2->mem
```
Fourth example: zero-copy forward

Modify the application from the previous example to support zero-copy if possible, and fallback to copy otherwise.
let us focus on the RX path of the NIC
we have seen that, when we open a NIC in netmap mode, the NIC is disconnected from the host stack
and netmap rings are allocated and pre-filled with netmap buffers
an additional, software-only TX netmap ring is also allocated and pre-filled with netmap buffers
this ring can be used to inject packets into the host stack, as if they were coming from the NIC
it can be used as any other netmap TX ring
it can be used as any other netmap TX ring
during sync, netmap will copy each new message in a new skbuf and pushes it up the stack
during sync, netmap will copy each new message in a new skbuf and pushes it up the stack
during sync, netmap will copy each new message in a new skbuf and pushes it up the stack
now let us consider the TX path of the NIC
when the NIC is open in netmap mode the stack TX is redirected to a software-only netmap RX ring
for netmap applications, it is an RX ring
when the stack wants to send packets through the NIC...
Netmap host rings (TX path)

...netmap intercepts the operations and copies the messages into the ring’s netmap buffers
Netmap host rings (TX path)

...netmap intercepts the operations and copies the messages into the ring’s netmap buffers
...netmap intercepts the operations and copies the messages into the ring's netmap buffers
after sync, the netmap application may then consume the messages
after sync, the netmap application may then consume the messages
after sync, the netmap application may then consume the messages
Let us consider a NIC with multiple TX rings
A successful call to `nm_open("netmap:eth0",...)` will redirect everything to the netmap RX host ring.
Instead, `nm_open("netmap:eth0-0", ...)` will redirect only TX ring 0.
Cards with multiple rings (TX path)

Similarly for `nm_open("netmap:eth0-1", ...)`

```
netmap host ring
  `network stack`
    `eth0`
```