

AF_XDP

Collins Huff

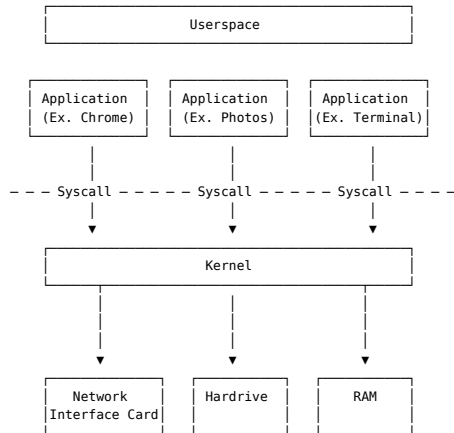
2021-06-15

Motivation

I'm interested in scanning the internet as fast as possible.

- ▶ There are 4,294,967,296 IPv4 Addresses
- ▶ Scanning all of IPv4 at 100,000 packets per second takes 12 hours
- ▶ Scanning all of IPv4 at 1,000,000 per second takes 71 minutes
- ▶ Scanning all of IPv4 at 10,000,000 packets per second takes 7 minutes

OS/Kernel Review



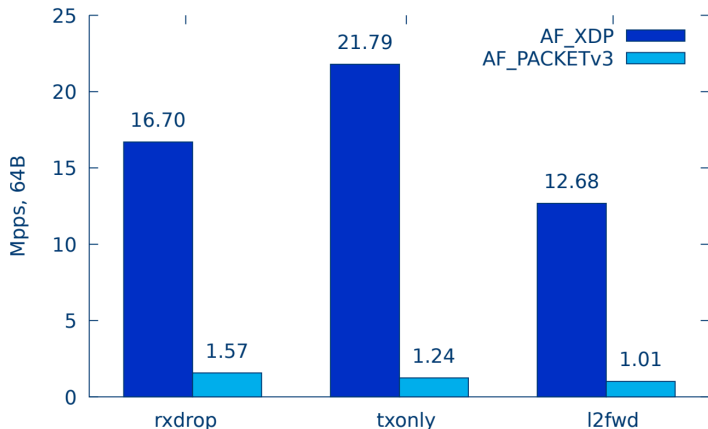
Fast Packet Processing

There are two main methods for fast packet processing:

- ▶ In-Kernel: AF_PACKET, in kernel, slow but easy to use
- ▶ Kernel Bypass (DPDK, Netmap, PF_RING), fast but hard to use

AF_XDP

AF_XDP is a third way: an in-kernel fast path. It is nearly as fast as kernel bypass, but it is built into the kernel.



Analogy

Imagine going to the airport

- ▶ In-Kernel packet processing is like going through TSA
- ▶ Kernel bypass is like showing up to the airport and getting on a private jet
- ▶ AF_XDP is like TSA Precheck

Applications

Applications in which you might need high performance packet processing:

- ▶ Intrusion Detection, Ex. Suricata
- ▶ L4 Load Balancing, Ex. Katran
- ▶ Quickly scanning the Internet, Ex. ZMap

How to Scan 0.0.0.0/0 - TCP

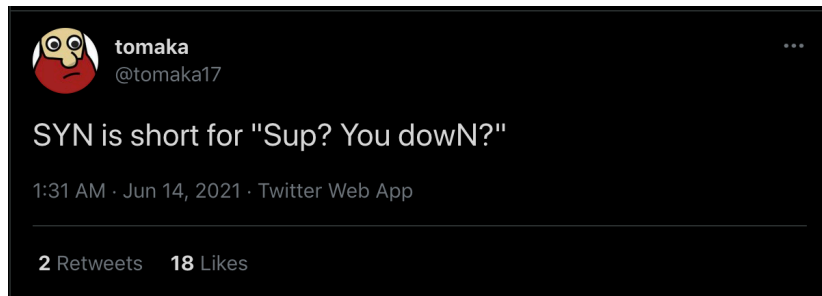
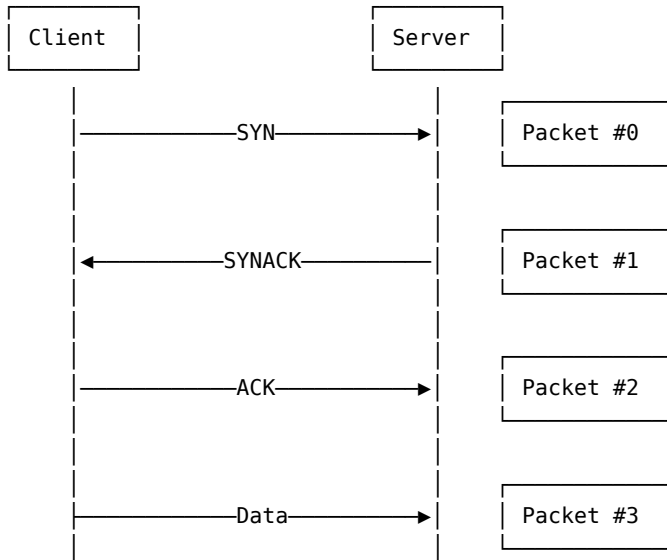


Figure 2: SYN

How to Scan 0.0.0.0/0



Zmap

Sends TCP SYN packets, listens for SYNACK to determine open ports.

Zmap

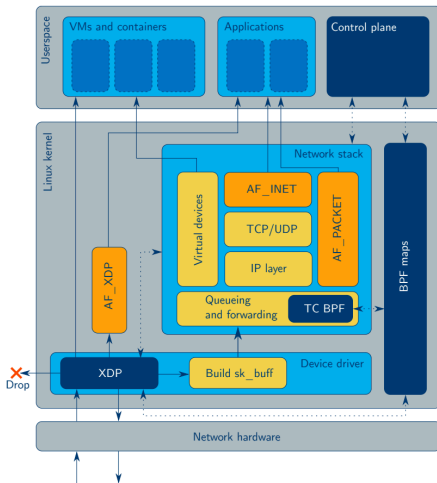
ZMap already provides high performance scanning using PF_RING.

However, to use PF_RING, you have to buy a license that costs \$150 per network interface.

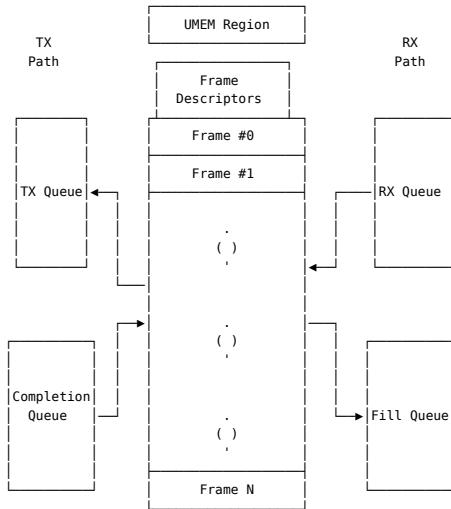
Since I'm too stingy to shell out for a PF_RING license, I set out to use AF_XDP to send packets with ZMap.

AF_XDP

AF_XDP is an address family that is optimized for high performance packet processing. AF_XDP is built on top of two layers of abstraction - eBPF - XDP



AF_XDP and xdpsock



Rewrite it in Rust

Starting point: <https://github.com/DouglasGray/xsk-rs>.

Similar to the `af_xdp` example in the kernel source tree.

Uses <https://github.com/alexforster/libbpf-sys>, which is used to set up the shared queues.

Issues

Two problems for my use case:

- ▶ Can't send and receive from multiple threads
- ▶ Complicated API

Design Issue

Original Design

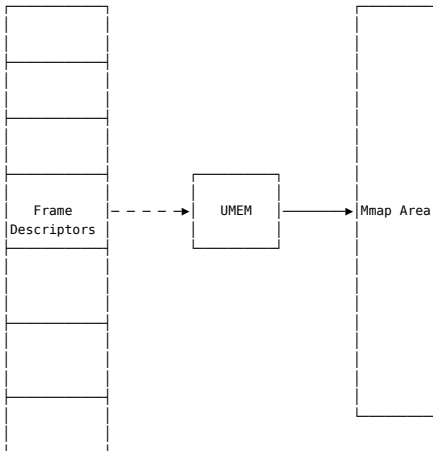
```
pub struct Umem<'a> {
    config: Config,
    frame_size: usize,
    umem_len: usize,
    mtu: usize,
    inner: Box<xsk_umem>,
    mmap_area: MmapArea,
    _marker: PhantomData<&'a ()>,
}

impl Umem<'a_> {
    pub unsafe fn read_from_umem(&self, addr: &usize, len: &usize) -
    > &[u8]

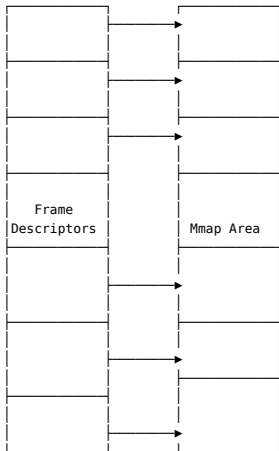
    pub unsafe fn write_to_umem(&mut self,
        frame_desc: &mut FrameDesc, data: &[u8])
}
```


Ownership Diagram

We can represent this with the following ownership diagram (Solid lines represent ownership, dashed lines represent references).



Revised Ownership Diagram



Unsafe Escape Hatch

```
pub struct Frame<'umem> {  
    addr: usize,  
    len: usize,  
    options: u32,  
    mtu: usize,  
    mmap_area: Arc<MmapArea>,  
    pub status: FrameStatus,  
}
```

Unsafe Escape Hatch

```
impl Frame {  
    ...  
    pub unsafe fn read_from_umem(&self, len: usize) -> &[u8] {  
        self.mmap_area.mem_range(self.addr, len)  
    }  
}
```

Unsafe Escape Hatch

...

```
pub unsafe fn write_to_umem(&mut self, data: &[u8]) {  
    let data_len = data.len();  
  
    if data_len > 0 {  
        let umem_region = self.mmap_area.mem_range_mut(&self.addr(), &data_len);  
        umem_region[..data_len].copy_from_slice(data);  
    }  
  
    self.set_len(data_len);  
}  
...  
}
```

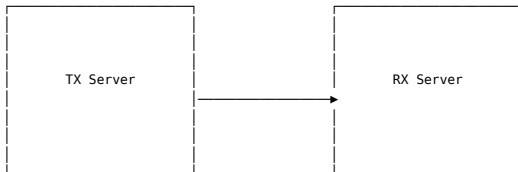
Simplifying the API

```
// Sending a packet
let pkt: Vec<u8> = vec![];
xsk.tx.send(&pkt);

// Receiving a packet
let pkt: Vec<u8> = vec![];
let len = xsk.recv(&mut pkt);
```

Performance Test Setup

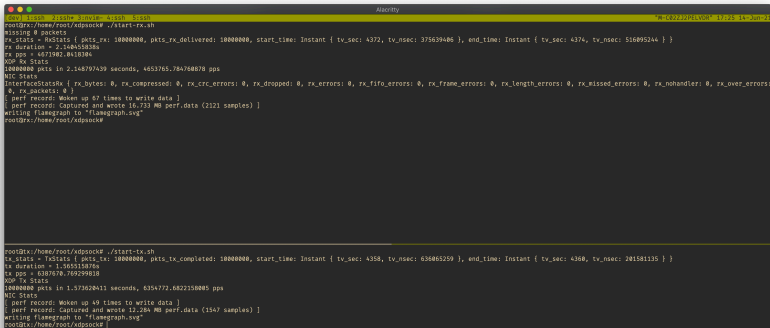
https://github.com/seeyarh/xdpsock/blob/master/examples/dev_to_dev.rs



Performance

Too slow

Should be able to get 14 million pps, only getting 5 million pps



```
Alacritty
"WM-C02220FELVDM" 37125 34x20pt-21
root@rx:/home/root/xdpsoc# ./start-rx.sh
missing 0 packets
tx_stats = TStats { pkts_tx: 10000000, pkts_rx_delivered: 10000000, start_time: Instant { tv_sec: 4372, tv_nsec: 375639406 }, end_time: Instant { tv_sec: 4374, tv_nsec: 516095244 } }
rx_duration = 2.148455038s
rx_pps = 4671982.0418304
XDP Rx Stats
10000000 pkts in 2.148797439 seconds, 4653765.784760878 pps
NIC Stats
InterfaceStatsRx { rx_bytes: 0, rx_compressed: 0, rx_crc_errors: 0, rx_dropped: 0, rx_errors: 0, rx_fifo_errors: 0, rx_frame_errors: 0, rx_length_errors: 0, rx_missed_errors: 0, rx_nohandler: 0, rx_over_errors: 0, rx_packets: 0 }
perf record: Woken up 67 times to write data ]
perf record: Captured and wrote 16.733 MB perf.data (2121 samples) ]
writing flamegraph to "flamegraph.svg"
root@rx:/home/root/xdpsoc#

root@rx:/home/root/xdpsoc# ./start-tx.sh
tx_stats = TStats { pkts_tx: 10000000, pkts_tx_completed: 10000000, start_time: Instant { tv_sec: 4358, tv_nsec: 636065259 }, end_time: Instant { tv_sec: 4360, tv_nsec: 201581135 } }
tx_duration = 1.56831878s
tx_pps = 6307670.769299818
XDP Tx Stats
10000000 pkts in 1.573678411 seconds, 6354772.0827350003 pps
NIC Stats
perf record: Woken up 49 times to write data ]
perf record: Captured and wrote 13.264 MB perf.data (1547 samples) ]
writing flamegraph to "flamegraph.svg"
root@rx:/home/root/xdpsoc#
```


Optimizing TX

Flamegraphs are a tool to visualize where your program is spending time. `cargo-flamegraph`

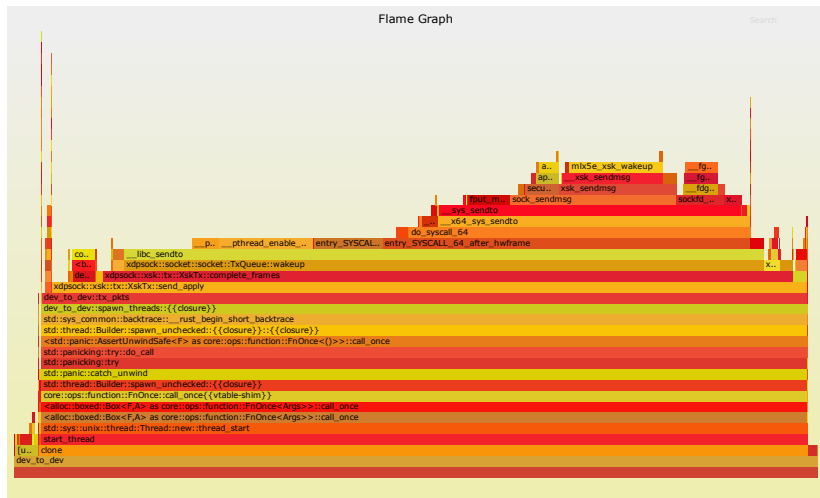


Figure 5: before

Send method unoptimized

The send method calls the complete frames method.

```
pub fn send(&mut self, data: &[u8])
    -> Result<(), XskSendError> {
    self.complete_frames();
    ...

    // Add consumed frames back to the tx queue
    if self.cur_batch_size == self.batch_size {
        self.put_batch_on_tx_queue();
    }

    Ok(())
}
```

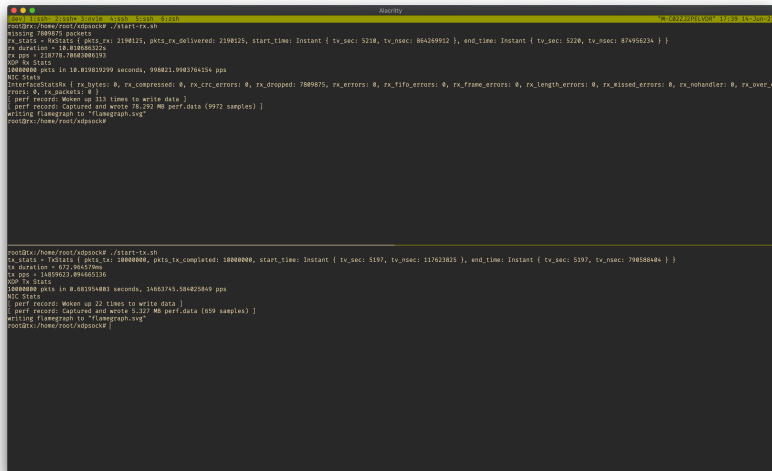
Send method unoptimized

```
fn put_batch_on_tx_queue(&mut self) {  
    ...  
    while unsafe {  
        self.tx_q  
            .produce_and_wakeup(&self.tx_frames[start..end])  
            .expect("failed to add frames to tx queue")  
    } != self.cur_batch_size  
    {  
        // Loop until frames added to the tx ring.  
    }  
    ...  
}
```

Send method unoptimized

```
/// Read frames from completion queue
fn complete_frames(&mut self) -> u64 {
    ...
    if n_free_frames == 0 {
        log::debug!("comp_q.consume() consumed 0 frames");
        if self.tx_q.needs_wakeup() {
            self.tx_q.wakeup()
                .expect("failed to wake up tx queue");
        }
    }
    ...
}
```

Optimizing TX



```
root@rx:/home/root@rx$ ./start-tx.sh
missing 7889875 packets
rx_stats = RxStats { pkts_rx: 2198125, pkts_tx_completed: 2198125, start_time: Instant { tv_sec: 5210, tv_nsec: 864269912 }, end_time: Instant { tv_sec: 5220, tv_nsec: 874956234 } }
rx_duration = 10.838086322s
rx_pos = 2187778.7880860193
XDP Tx Stats
10000000 pkts in 10.83810290 seconds, 908021.9983784154 pps
NIC Stats
InterfaceStatsRx { rx_bytes: 0, rx_compressed: 0, rx_crc_errors: 0, rx_dropped: 7889875, rx_errors: 0, rx_fifo_errors: 0, rx_frame_errors: 0, rx_length_errors: 0, rx_missed_errors: 0, rx_nohandler: 0, rx_over_e
rrors: 0, rx_packets: 0 }
[ perf record: Woken up 313 times to write data ]
[ perf record: Captured and wrote 78.292 MB perf.data (9972 samples) ]
writing flamegraph to "flamegraph.svg"
root@rx:/home/root@rx$

root@rx:/home/root@rx$ ./start-tx.sh
rx_stats = TxStats { pkts_tx: 10000000, pkts_tx_completed: 10000000, start_time: Instant { tv_sec: 5197, tv_nsec: 117623825 }, end_time: Instant { tv_sec: 5197, tv_nsec: 798588044 } }
rx_duration = 0.727664579ms
tx_pos = 14889623.89465136
XDP Tx Stats
10000000 pkts in 0.687954083 seconds, 14683745.584825849 pps
NIC Stats
[ perf record: Woken up 22 times to write data ]
[ perf record: Captured and wrote 0.327 MB perf.data (659 samples) ]
writing flamegraph to "flamegraph.svg"
root@rx:/home/root@rx$
```

Figure 6: after

Optimizing TX

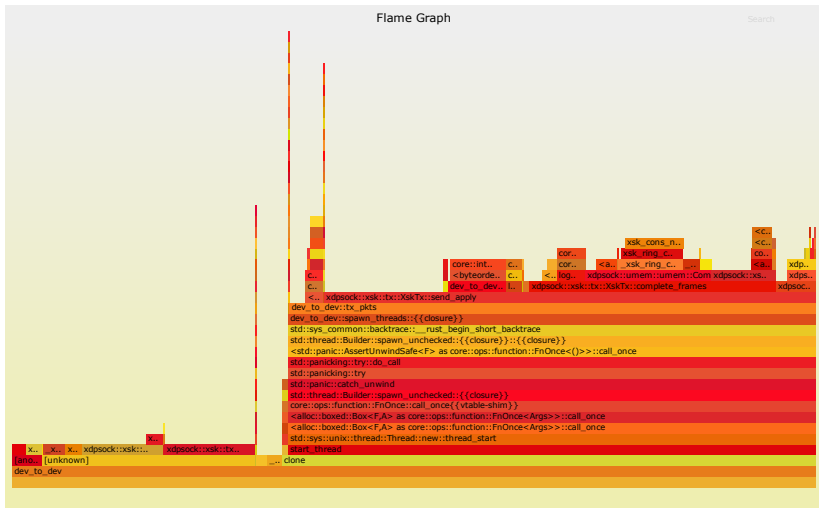
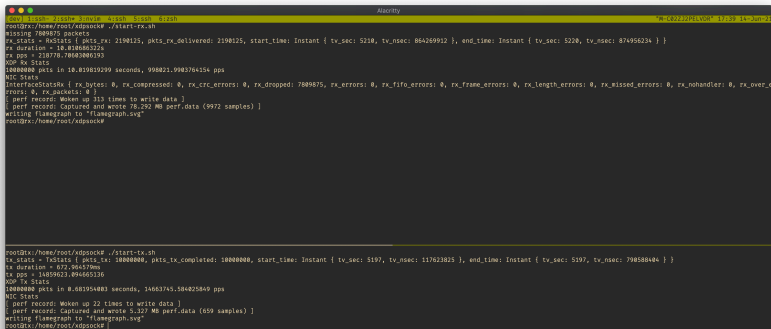


Figure 7: after

Optimizing RX

Now that we have optimized the TX path, we have a new problem: the RX path can't keep up.

We are missing 7,809,875 packets out of 10,000,000 packets, or 78%.



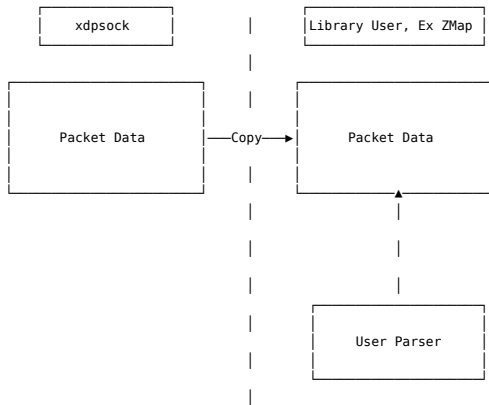
```
root@rx:/home/root/xdpsoc# ./start-rx.sh
missing 7809875 packets
rx_stats = RxStats { pkts_rx: 2198125, pkts_rx_delivered: 2198125, start_time: Instant { tv_sec: 5210, tv_nsec: 864269912 }, end_time: Instant { tv_sec: 5220, tv_nsec: 874956234 } }
rx_duration = 10.020806327s
rx_pps = 218776.7000000193
UDP Rx Stats
10000000 pkts in 10.020819299 seconds, 998021.9903764154 pps
NIC Stats
InterfaceStatsRx { rx_bytes: 0, rx_compressed: 0, rx_crc_errors: 0, rx_dropped: 7809875, rx_errors: 0, rx_fifo_errors: 0, rx_frame_errors: 0, rx_length_errors: 0, rx_missed_errors: 0, rx_nohandler: 0, rx_over_e
rrors: 0, rx_packets: 0 }
perf record: Woken up 313 times to write data
perf record: Captured and wrote 78.292 MB perf.data (9972 samples)
writing flamegraph to "flamegraph.svg"
root@rx:/home/root/xdpsoc#

root@rx:/home/root/xdpsoc# ./start-tx.sh
tx_stats = TxStats { pkts_tx: 10000000, pkts_tx_completed: 10000000, start_time: Instant { tv_sec: 5197, tv_nsec: 117623825 }, end_time: Instant { tv_sec: 5197, tv_nsec: 798588404 } }
tx_duration = 0.72964579ms
tx_pps = 14059673.894665136
UDP Tx Stats
10000000 pkts in 0.687954083 seconds, 14663743.584025849 pps
NIC Stats
perf record: Woken up 22 times to write data
perf record: Captured and wrote 0.327 MB perf.data (659 samples)
writing flamegraph to "flamegraph.svg"
root@rx:/home/root/xdpsoc#
```

Optimizing RX

```
pub fn recv(&mut self, pkt_receiver: &mut [u8]) -> usize {
```

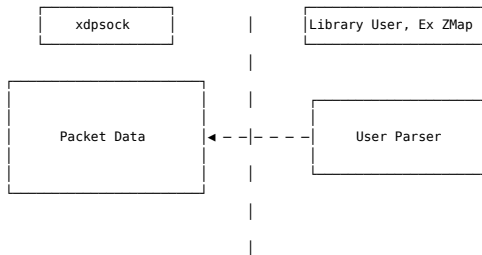

Optimizing RX - Copy



Optimizing RX - Zerocopy

Accept a function, use a closure

Optimizing RX - Zerocopy



Optimizing RX: avoiding copies

```
pub fn recv_apply<F>(&mut self, f: F)
where
    F: FnMut(&[u8]),
{
    ...
    if n_frames_recv > 0 {
        self.apply_batch(n_frames_recv, f);
    }
    ...
}
```

Optimizing RX: avoiding copies

```
fn apply_batch<F>(&mut self, n_frames_rcv: usize, mut f: F)
where
    F: FnMut(&[u8]),
{
    ...

    for filled_frame in filled_frames {

        let data = unsafe { filled_frame.read_from_umem(frame.len()) };
        f(data);
    }
    ...
}
```

Optimizing RX: avoiding copies

Now we are only missing 403,862 packets out of 10,000,000 packets, or 4%.

```
root@rx:/home/root# ./start-rx.sh
missing 403862 packets
rx_stats = RxStats { pkts_rx: 9596138, pkts_rx_delivered: 9596138, start_time: Instant { tv_sec: 5453, tv_nsec: 254987839 }, end_time: Instant { tv_sec: 5463, tv_nsec: 267714482 } }
rx duration = 10.022087443s
rx pps = 959386.332162296
NOP Rx Stats
10000000 pkts in 10.025371474 seconds, 997469.2734263465 pps
NIC Stats
InterfaceStatsRx { rx_bytes: 0, rx_compressed: 0, rx_crc_errors: 0, rx_dropped: 403862, rx_errors: 0, rx_fifo_errors: 0, rx_frame_errors: 0, rx_length_errors: 0, rx_missed_errors: 0, rx_nohandler: 0, rx_over_er
rors: 0, rx_packets: 0 }
root@rx:/home/root# ./start-rx.sh

root@rx:/home/root# ./start-tx.sh
tx_stats = TxStats { pkts_tx: 10000000, pkts_tx_completed: 10000000, start_time: Instant { tv_sec: 5439, tv_nsec: 739795926 }, end_time: Instant { tv_sec: 5440, tv_nsec: 422387106 } }
tx duration = 0.7259178ss
tx pps = 14067872.63549827
NOP Tx Stats
10000000 pkts in 0.68853917 seconds, 14694231.340129916 pps
NIC Stats
[ perf record: Woken up 22 times to write data ]
[ perf record: Captured and wrote 0.343 MB perf.data (661 samples) ]
writing flamegraph to "flamegraph.svg"
root@rx:/home/root# ./start-rx.sh
```

C FFI

The Rust FFI Omnibus

C FFI

```
#[no_mangle]
pub extern "C" fn xsk_new<'a>(ifname: *const c_char) -
> *mut Xsk2<'a> {
    let ifname = unsafe {
        assert!(!ifname.is_null());
        CStr::from_ptr(ifname)
    };

    let ifname = ifname.to_str().unwrap();

    let mut xsk = Xsk2::new(&ifname, 0,
        umem_config, socket_config, n_tx_frames as usize);

    Box::into_raw(Box::new(xsk))
}
```


C FFI

```
#[no_mangle]
pub extern "C" fn xsk_send(xsk_ptr: *mut Xsk2,
    pkt: *const u8, len: size_t) {

    let xsk = unsafe {
        assert!(!xsk_ptr.is_null());
        &mut *xsk_ptr
    };

    let pkt = unsafe {
        assert!(!pkt.is_null());
        slice::from_raw_parts(pkt, len as usize)
    };

    xsk.send(&pkt);
}
```

C FFI

```
#[no_mangle]
pub extern "C" fn xsk_recv(xsk_ptr: *mut Xsk2,
    pkt: *mut u8, len: size_t) {

    let xsk = unsafe {
        assert(!xsk_ptr.is_null());
        &mut *xsk_ptr
    };

    let mut pkt = unsafe {
        assert(!pkt.is_null());
        slice::from_raw_parts_mut(pkt, len as usize)
    };

    let (recvd_pkt, len) = xsk.recv().expect("failed to recv");
    pkt[..len].clone_from_slice(&recvd_pkt[..len]);
}
```

C FFI

```
char* ifname = "veth0";  
void* xsk = xsk_new(ifname);  
...  
  
for(i = 0; i < pkts_to_recv; i++) {  
    char buf[MAX_PKT_SIZE] = {0};  
    xsk_recv(xsk, &buf, len);  
}  
  
...  
  
for(i = 0; i < pkts_to_send; i++) {  
    xsk_send(xsk, &pkt_to_send, 50);  
}  
  
...  
xsk_delete(xsk);
```