# AF_XDP

Collins Huff

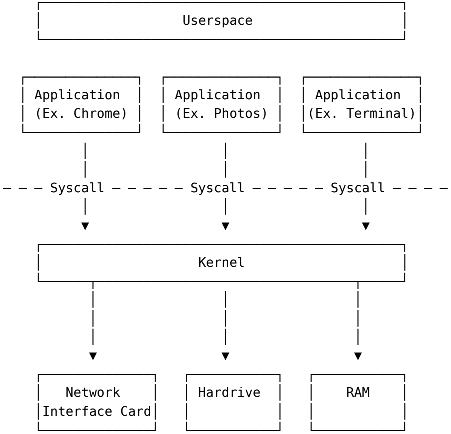
2021-06-15

## 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



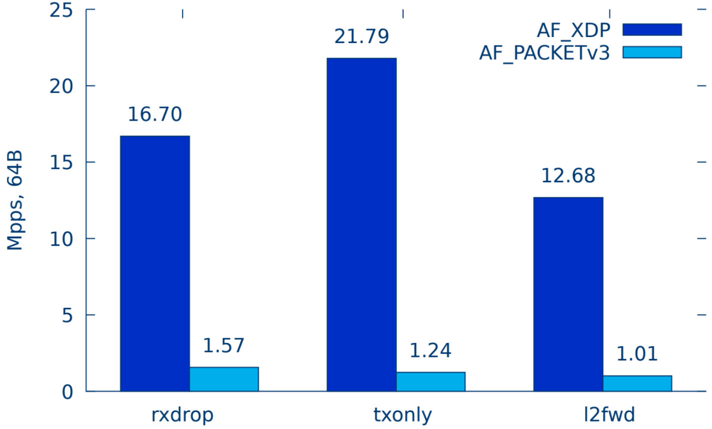
## Fast Packet Processing

The 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

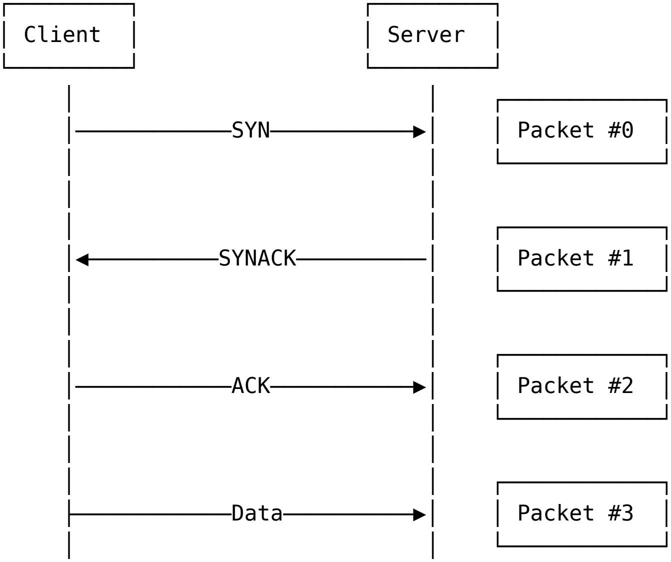
- In-Kernel packet processing is like going through TSA
- Kernel bypass is like showing up to the airport and getting on a private jet
- AF_XDP is like TSA Precheck

## 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

## Zmap



Client
Server
SYN
Packet #0
SYNACK
Packet #1
ACK
Packet #2
Data
Packet #3

## Zmap

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

## Zmap

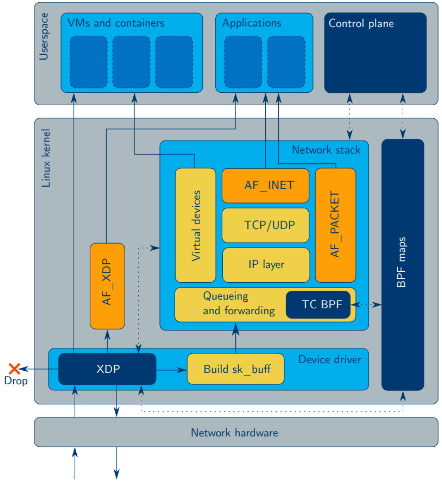
ZMap already provides high performance scanning using PF_RING.

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

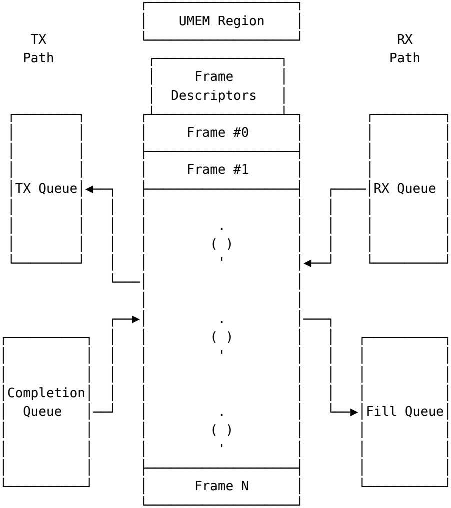
Since I’m too stingy to shell out for a PF_RING license, I set out to use AF_XDP to send packets with ZMap.

## 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

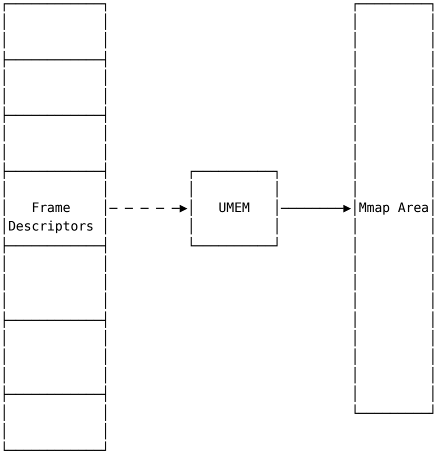
## Design Issue

### Original Design

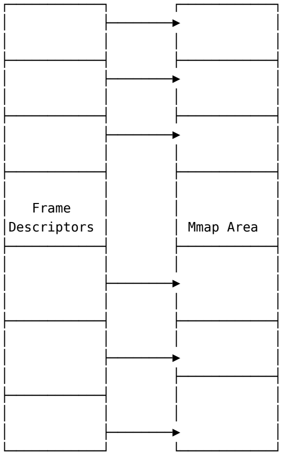
```
pub struct Umem<'a> {
    config: Config,
    frame_size: usize,
    umem_len: usize,
    mtu: usize,
    inner: Box<xsk_umem>,
    mmap_area: MmapArea,
    _marker: PhantomData<&'a ()>,
}

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

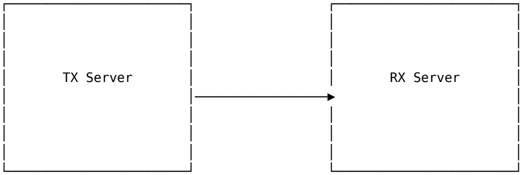
```
...

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(), &dat

        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

# Optimizing TX

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

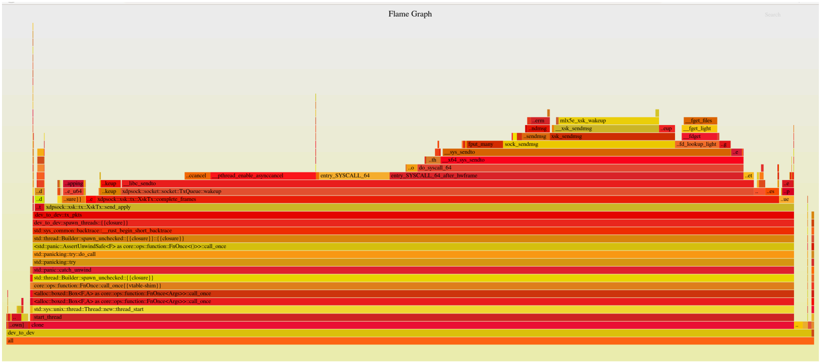


Figure 3: before

## 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

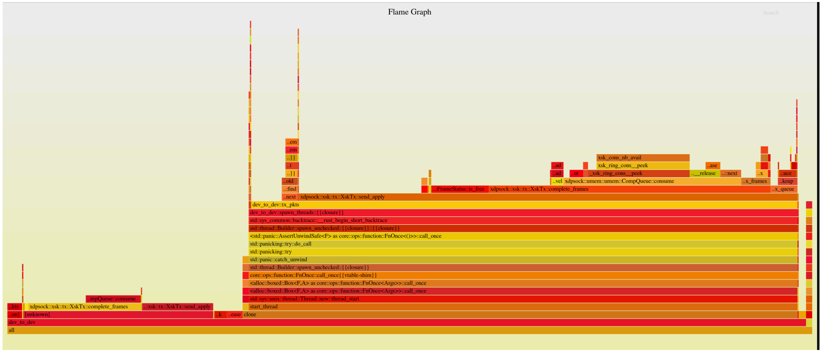


Figure 4: after

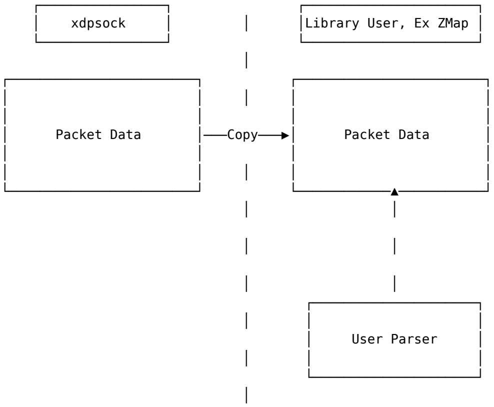
## Optimizing RX

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

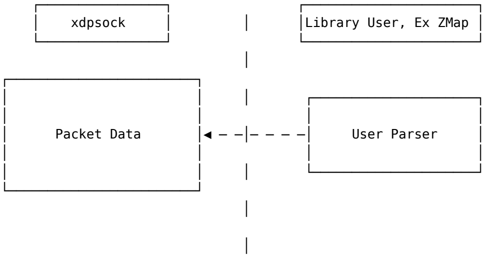
## Optimizing RX

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

# Optimizing RX - Copy



## 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_recv: 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);
    }
...
}
```

## 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);
```