AF_XDP

Collins Huff

2021-06-15

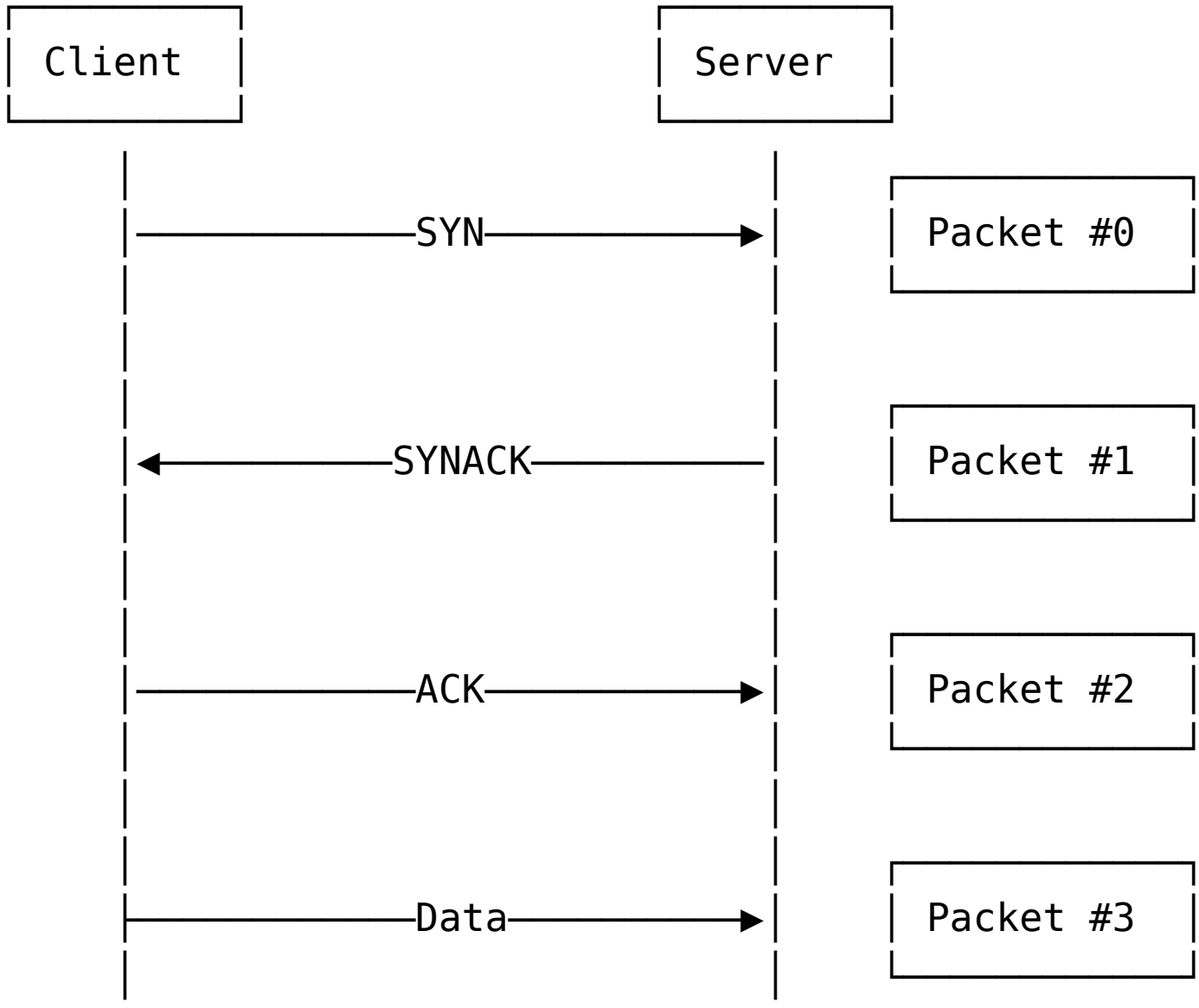
Motivation

I’m interested in scanning the internet.

How to Scan 0.0.0.0/0 - TCP

Figure 1: SYN

How to Scan 0.0.0.0/0 - TCP



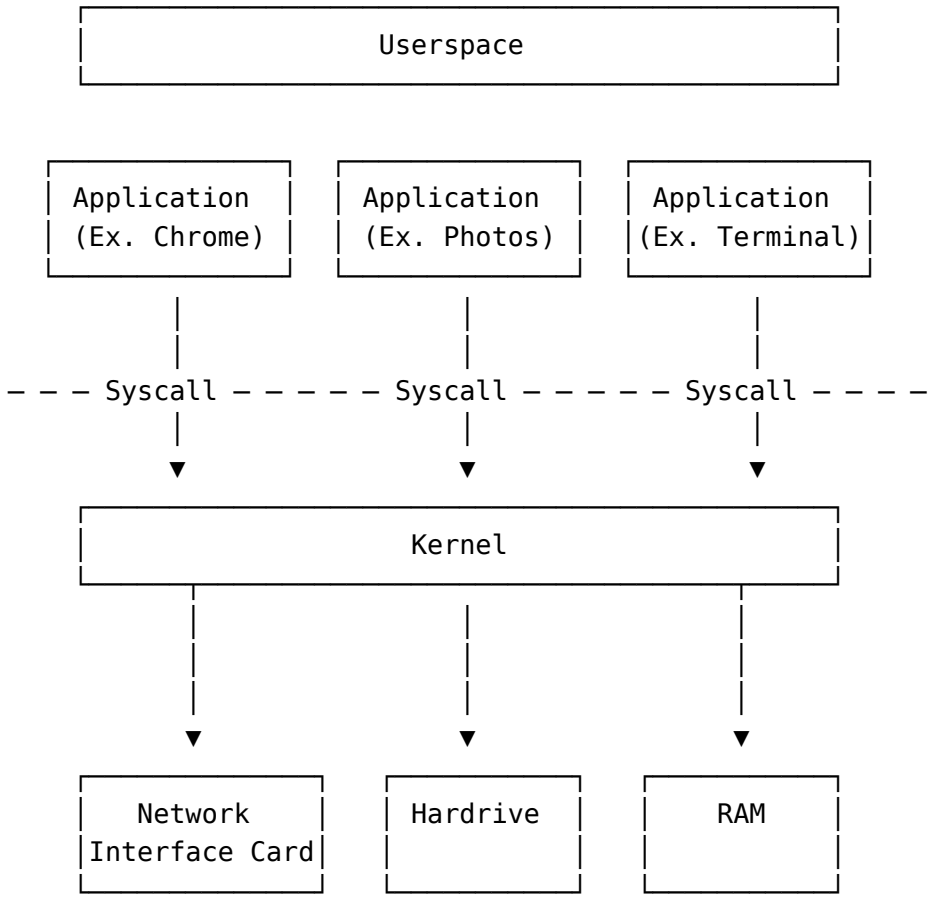
Zmap

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

Speed Matters

- 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:

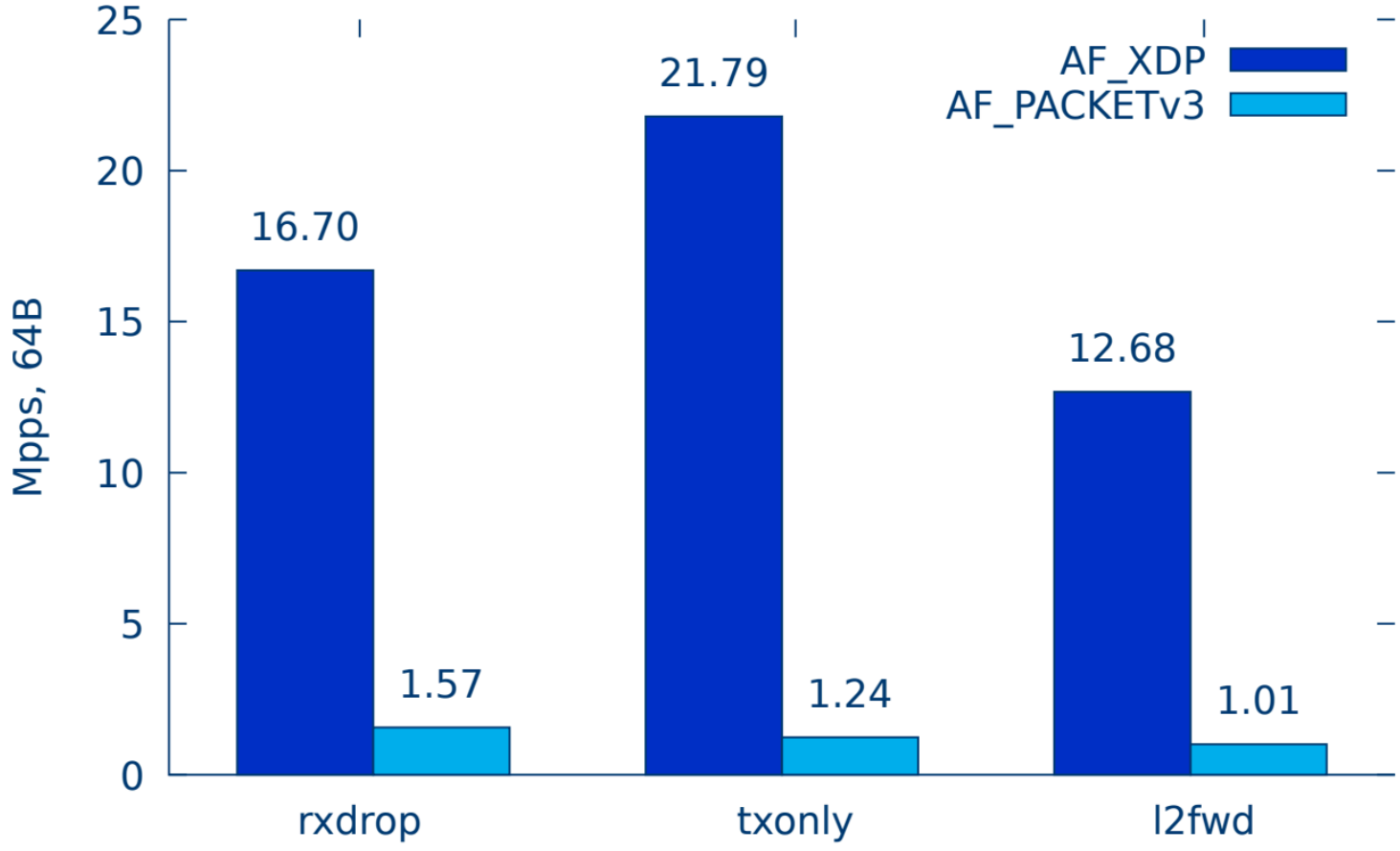
- In-Kernel: AF_PACKET, in kernel, slow but easy to use
- Kernel Bypass (DPDK, Netmap, PF_RING), fast but hard to use

Zmap

- AF_PACKET by default
- PF_RING if you buy a license

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.



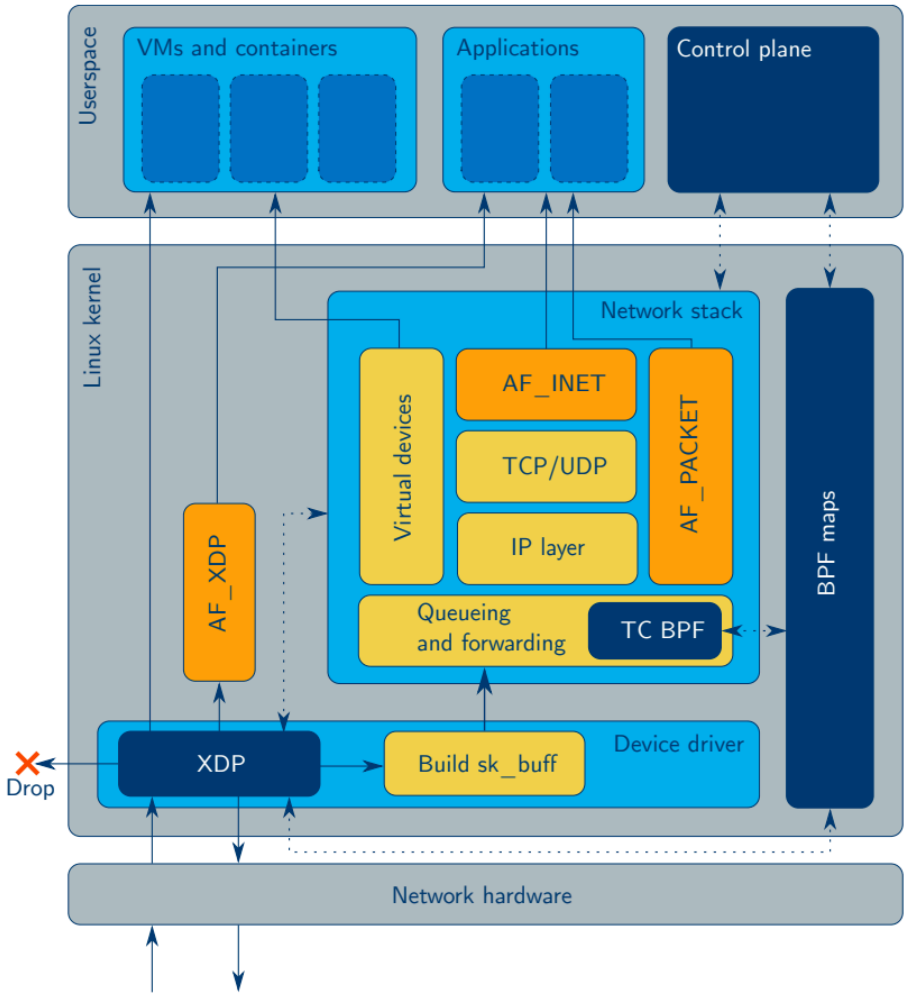
Analogy

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

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 components

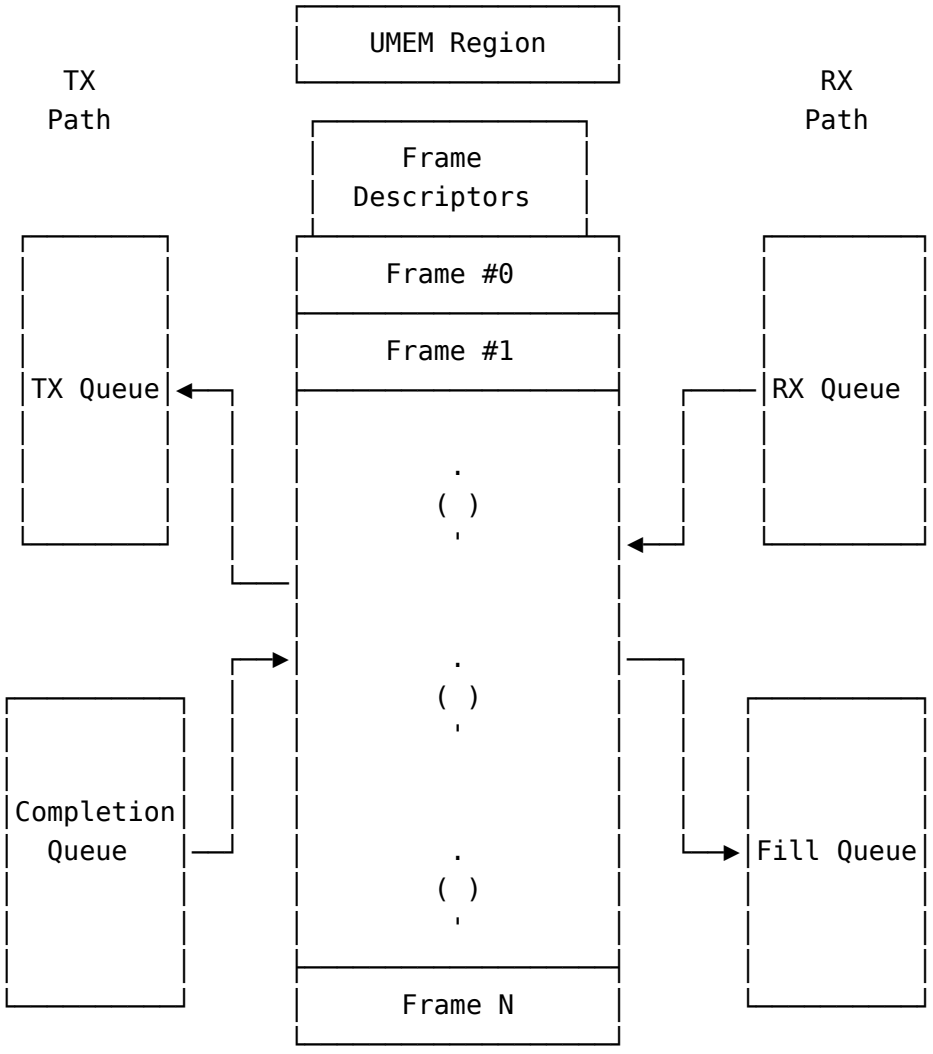
- 2 queues for TX: the TX Queue and Completion Queue
- 2 queues for RX: the RX Queue and Fill Queue
- 1 region of memory called the UMEM, shared between userspace and the kernel.

UMEM and MMAP

We need to allocate a big block of memory to use AF_XDP. The best way to do this is with mmap.

https://man7.org/linux/man-pages/man2/mmap.2.html

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

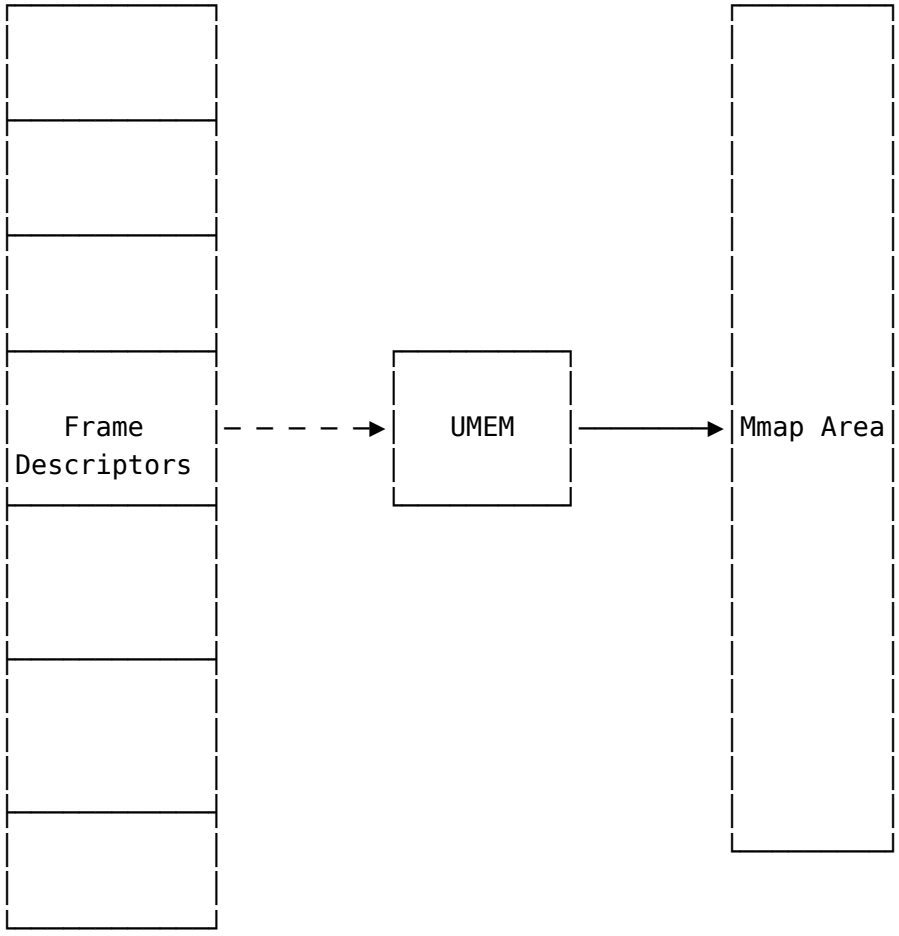
Design Issue

Original Design

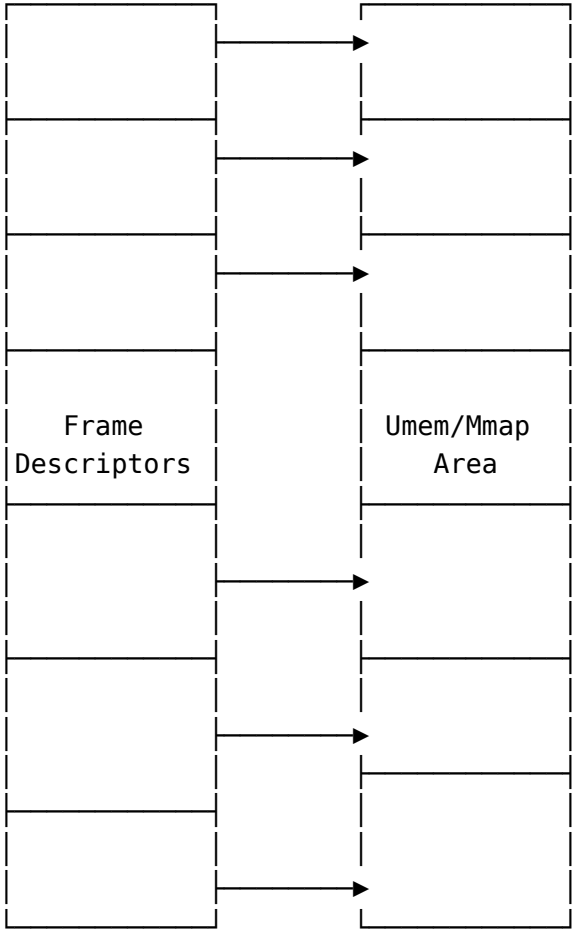
```
pub struct Umem<'a> {
    config: Config,
    frame_size: usize,
    umem_len: usize,
    mtu: usize,
    inner: Box<xsk_umem>,
    mmap_area: MmapArea,
    _marker: PhantomData<&'a ()>,
}

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

Each frame holds an Arc to the Umem region and constructs it's corresponding slice of bytes using a call to `slice::from_raw_parts_mut`.

Unsafe

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

Unsafe

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

Unsafe

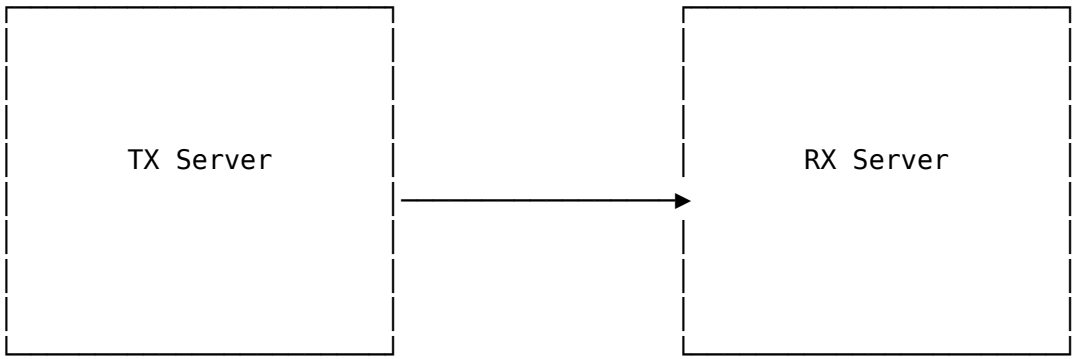
```
...

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

Performance Test Setup

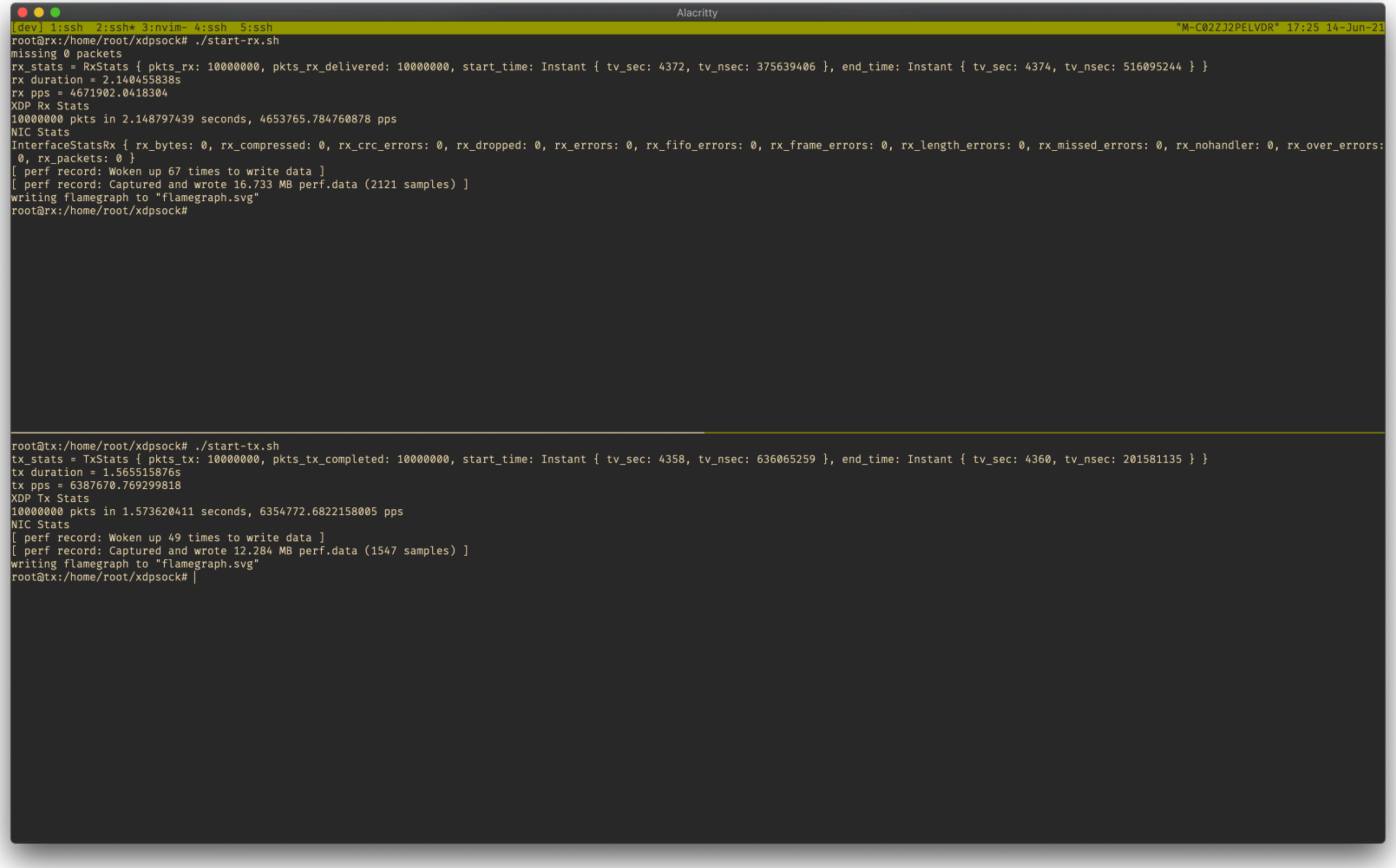
https://github.com/seeyarh/xdpsock/blob/master/examples/dev_to_dev.rs



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

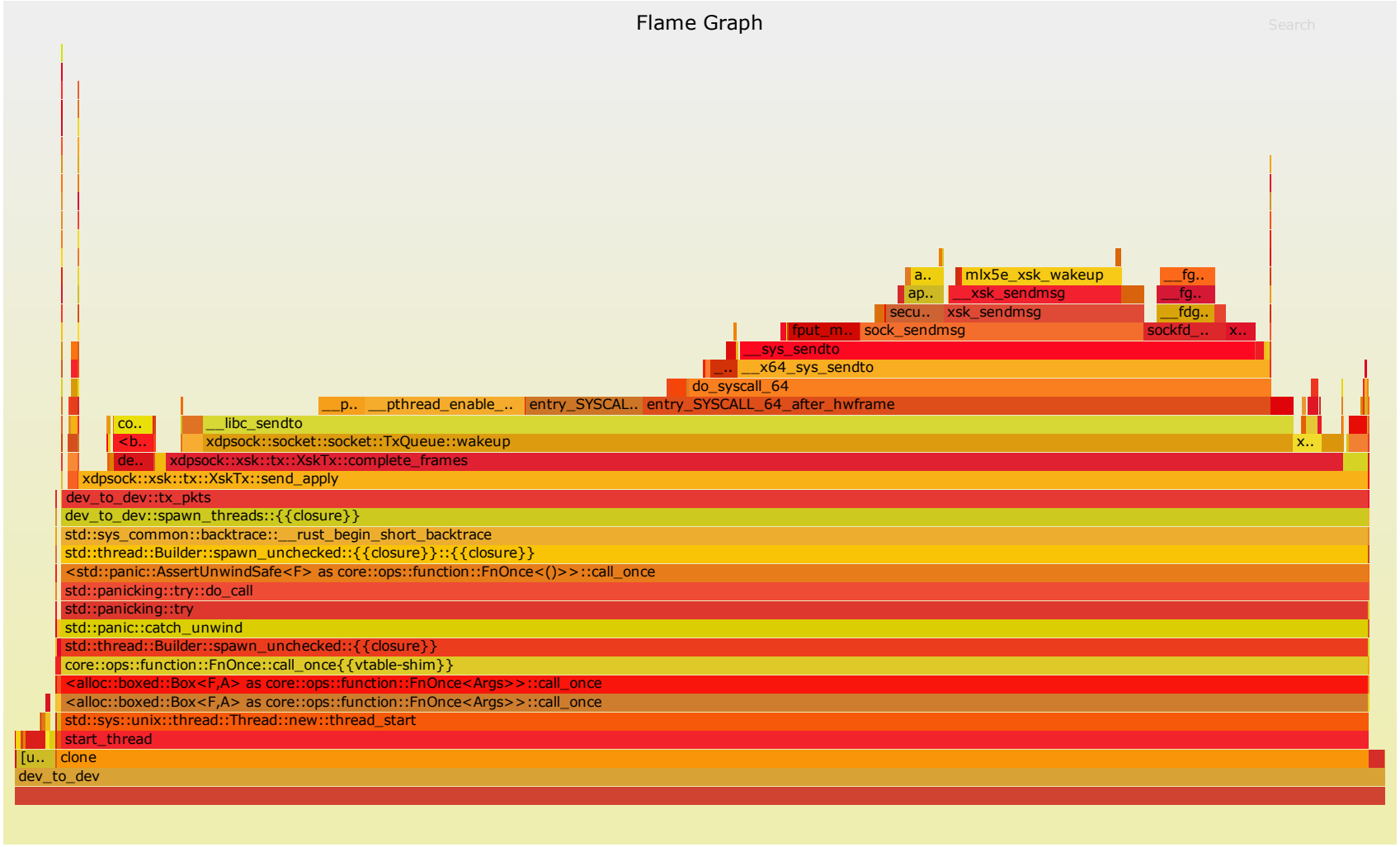


Figure 5: before

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

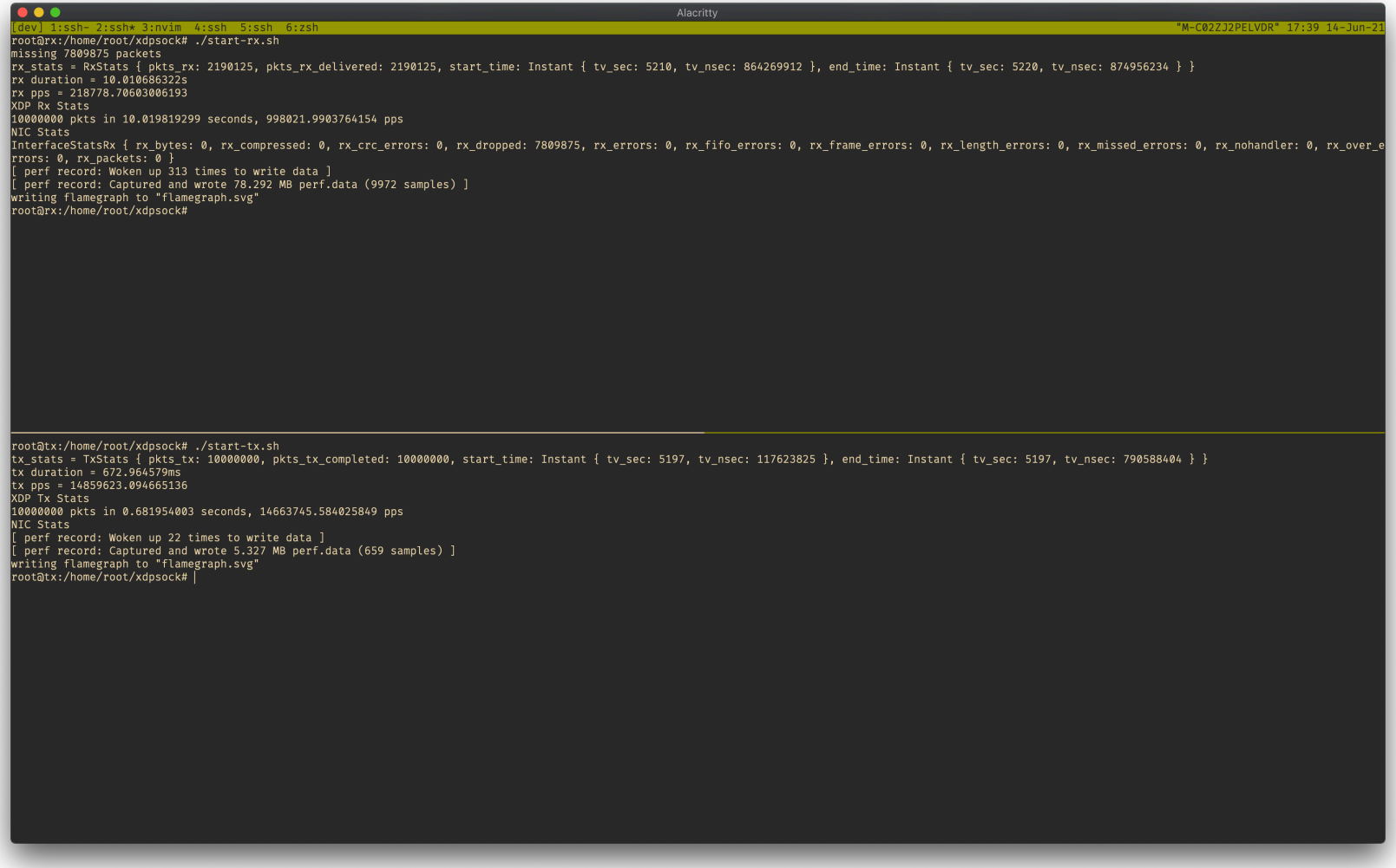


Figure 6: after

Optimizing TX

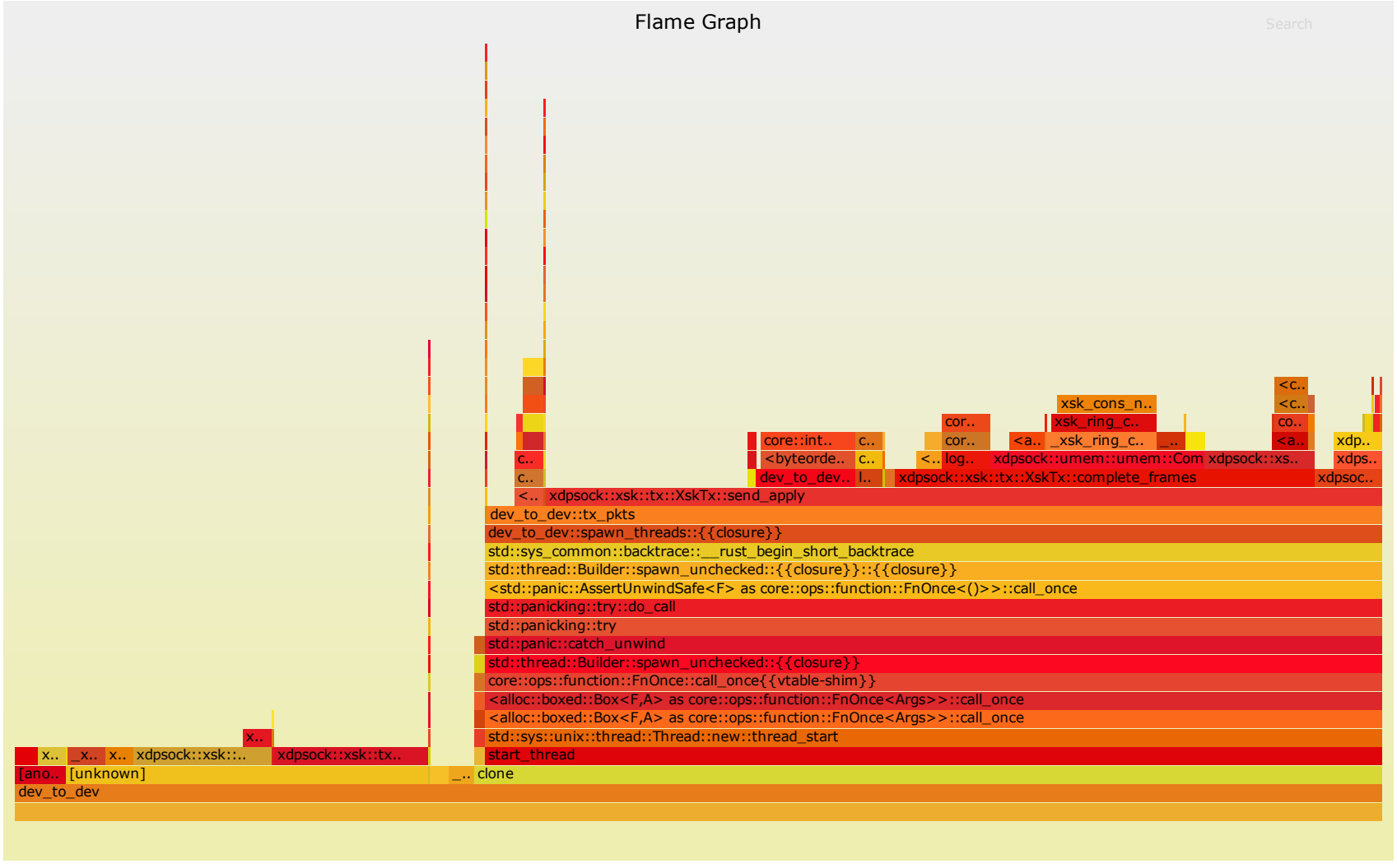
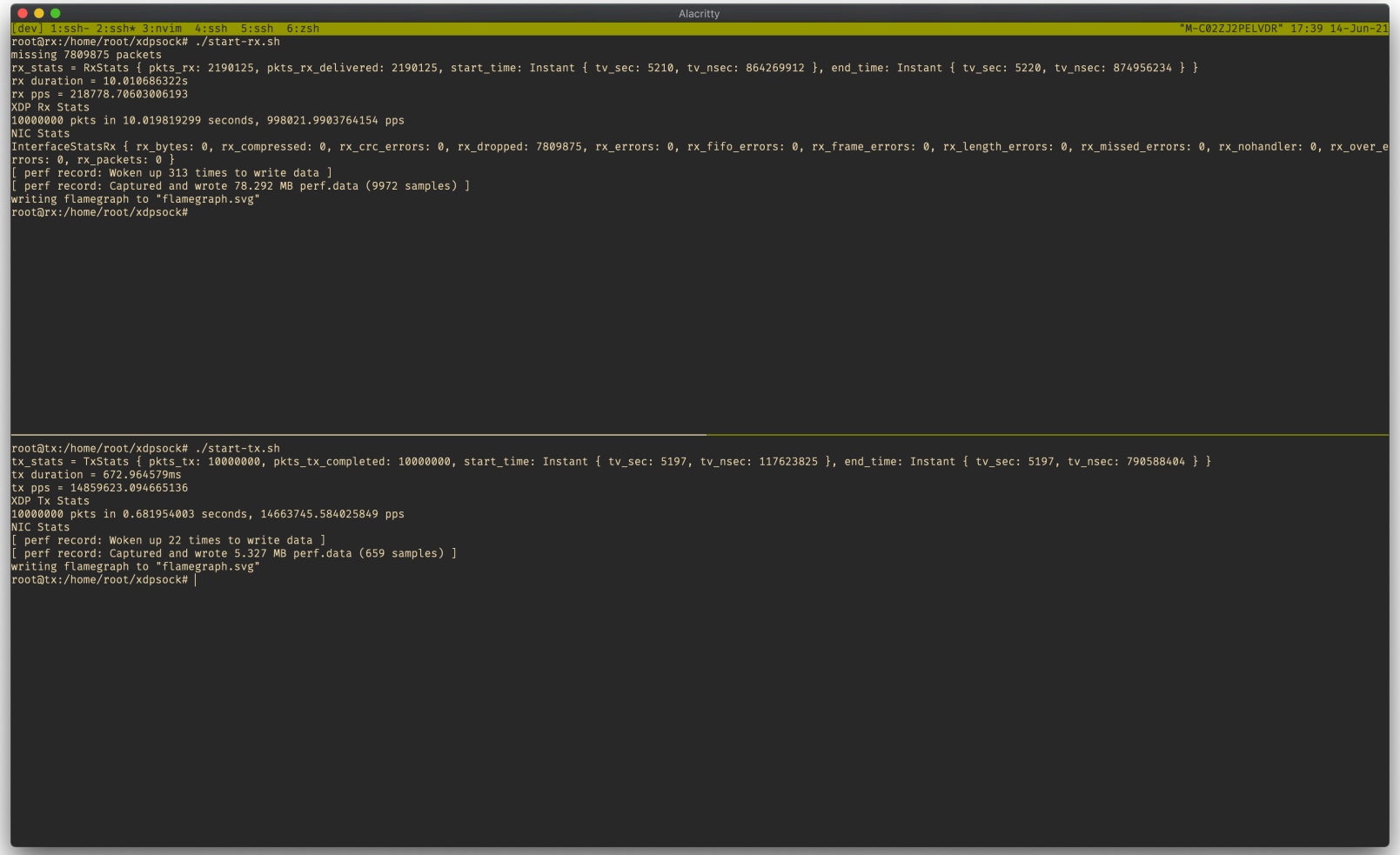


Figure 7: after

Optimizing RX

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

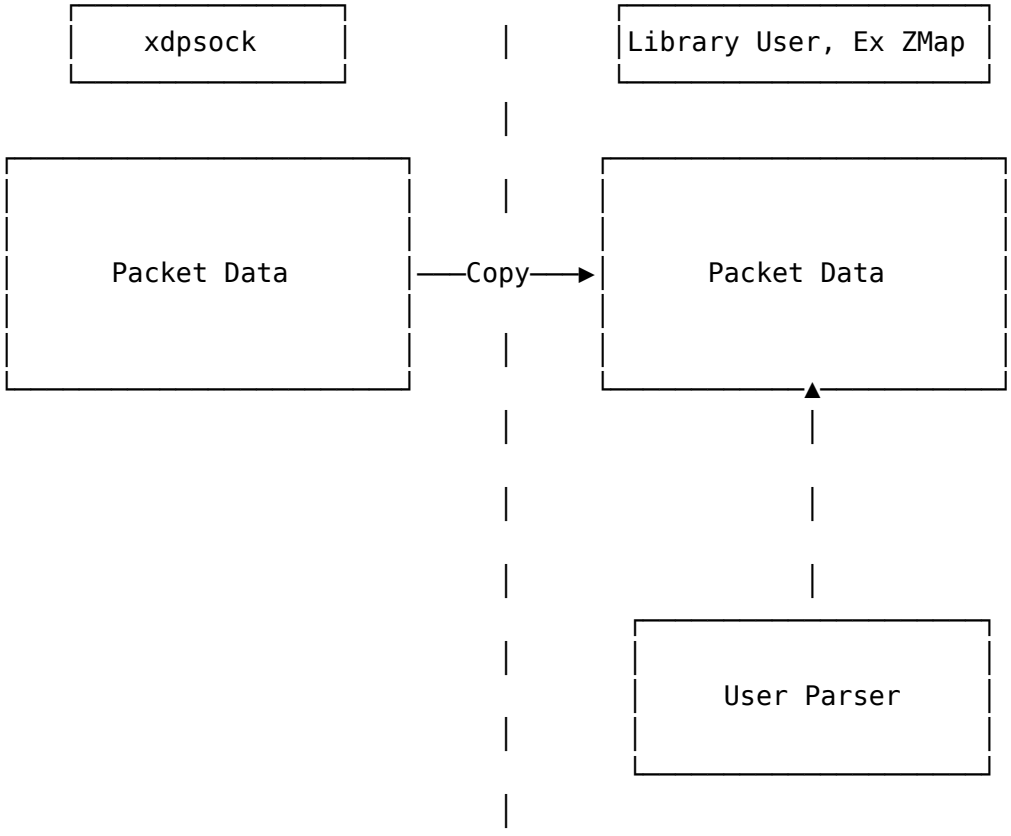
We are missing 7,809,875 packets out of 10,000,000 packets, or 78%.



Optimizing RX

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

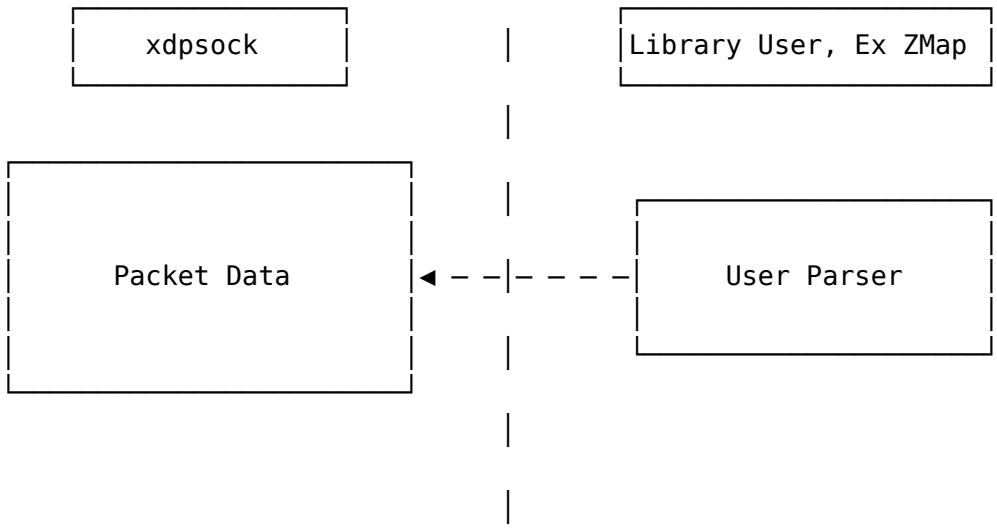
Optimizing RX - Copy



Optimizing RX - Zerocopy

Accept a function, use a closure

Optimizing RX - Zerocopy



Optimizing RX: avoiding copies

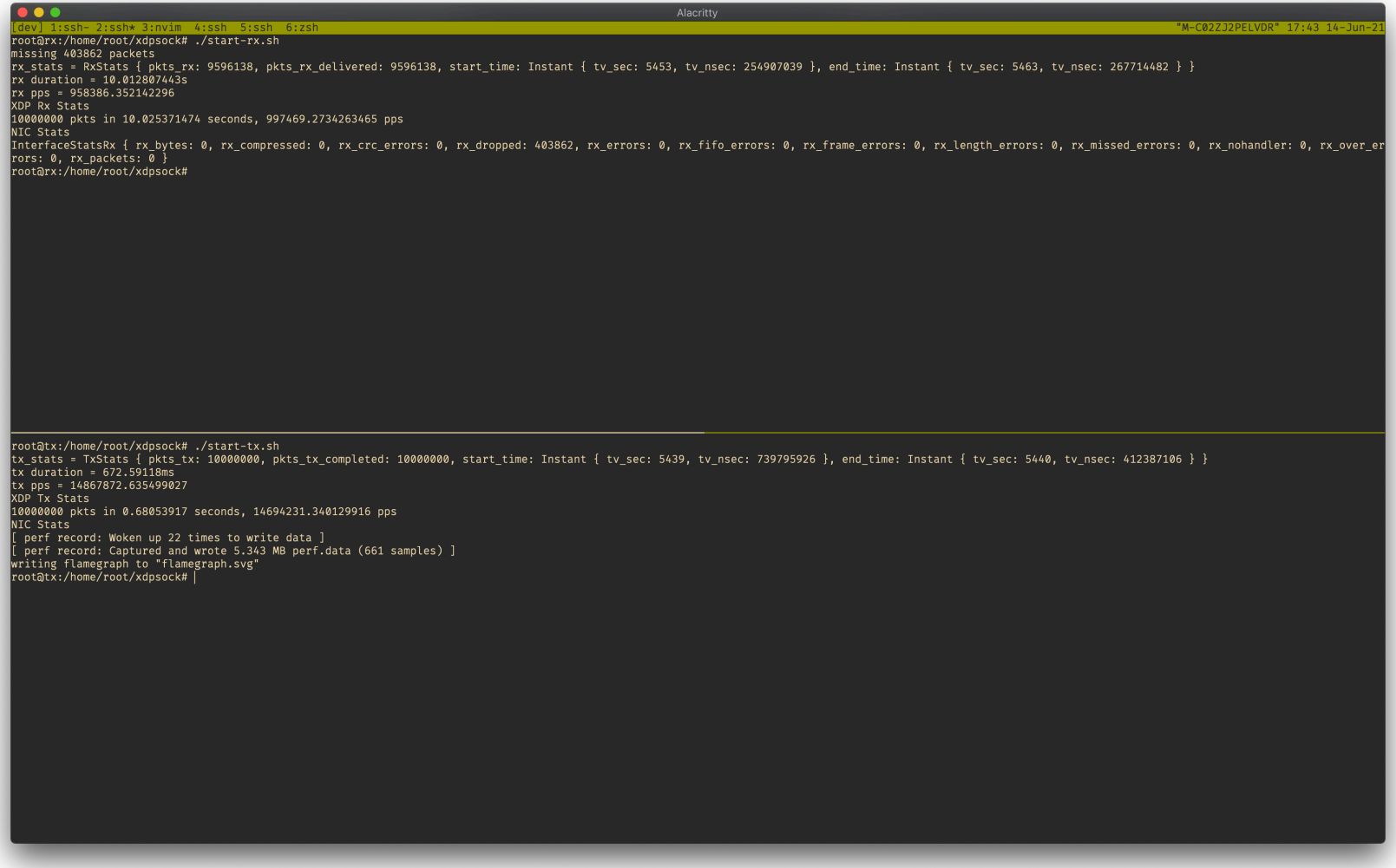
```
pub fn recv_apply<F>(&mut self, f: F)
where
    F: FnMut(&[u8]),
{
...
    if n_frames_recv > 0 {
        self.apply_batch(n_frames_recv, f);
    }
...
}
```

Optimizing RX: avoiding copies

```
fn apply_batch<F>(&mut self, n_frames_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);
    }
...
}
```

Optimizing RX: avoiding copies

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



C FFI

The Rust FFI Omnibus

http://jakegoulding.com/rust-ffi-omnibus/

C FFI

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

    let ifname = ifname.to_str().unwrap();
    let umem_config = UmemConfigBuilder::new()
        ...
    let socket_config = SocketConfigBuilder::new()
        ...
    let n_tx_frames = umem_config.frame_count() / 2;
    let n_tx_batch_size = 1024;

    let xsk = Xsk2::new(
        &ifname,
        0,
        umem_config,
        socket_config,
        n_tx_frames as usize,
        n_tx_batch_size,
    )
    .expect("failed to build xsk");
    Box::into_raw(Box::new(xsk))
}
```

C FFI

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

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

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

    xsk.tx.send(&pkt).expect("failed to send pkt");
}
```

C FFI

```
#[no_mangle]
pub unsafe extern "C" fn xsk_recv(xsk_ptr: *mut Xsk2,
    pkt: *mut u8, len: size_t) -> u16 {

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

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

    xsk.rx.recv(pkt) as u16
}
```

C FFI

```
int main() {
    char* ifname = "veth0";
    void* xsk = xsk_new(ifname);
    uint16_t len_recvd;

    int i, j;
    int pkts_to_recv = 10;
    size_t len = 1500;


    for(i = 0; i < pkts_to_recv; i++) {
        char buf[MAX_PKT_SIZE] = {0};
        len_recvd = xsk_recv(xsk, &buf, len);
        for(j = 0; j < len_recvd; j++) {
            printf("0x%hhx,", buf[j]);
        }
        printf("\n");
    }

    char pkt_to_send[50] = {...};


    for(i = 0; i < pkts_to_recv; i++) {
        xsk_send(xsk, &pkt_to_send, 50);
    }


    xsk_delete(xsk);
    return 0;
}
```

C FFI

https://github.com/seeyarh/zmap/tree/feature/af_xdp