ERIK PELLIZZON

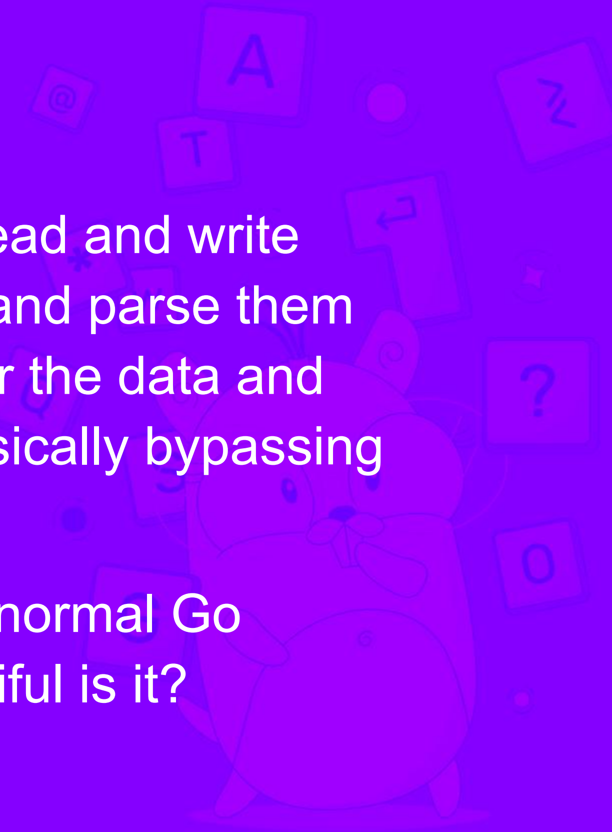
Go Developer - Freelancer

BYPASSING THE LINUX NET STACK WITH GO

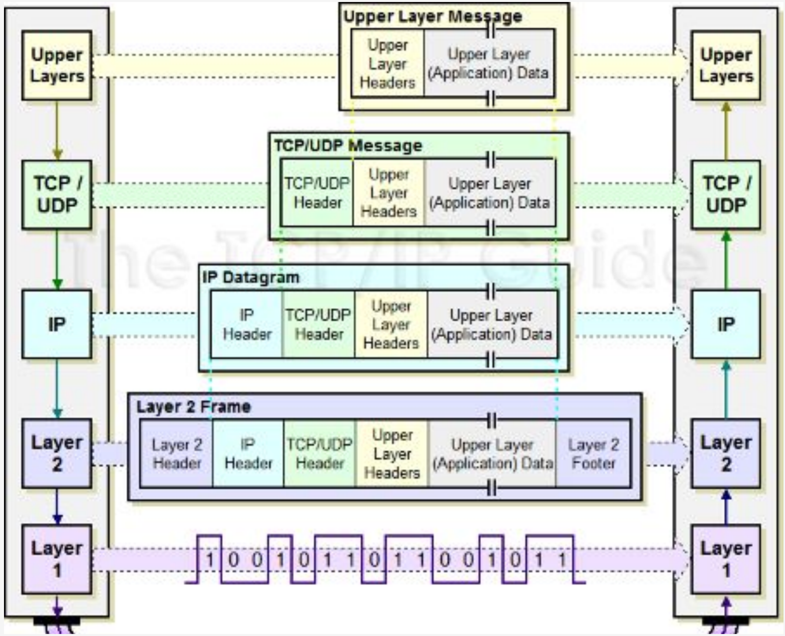
Introduction

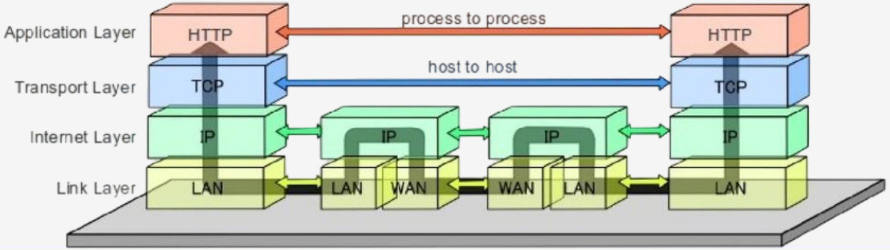
Using XDP and eBPF, we can directly read and write Ethernet frames from the network card and parse them ourselves, gaining complete control over the data and significantly improving performance, basically bypassing all the Linux network layers.

And all this can be done directly from a normal Go application run by the user... How beautiful is it?



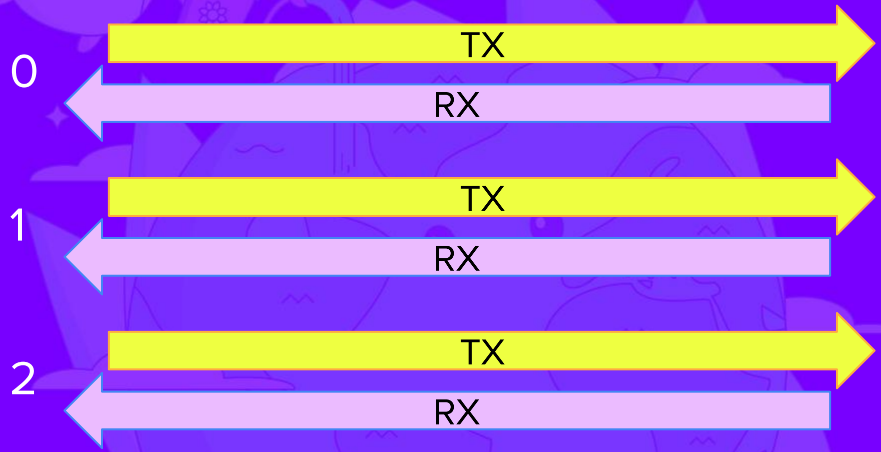
Network


Upper Layers
TCP / UDP
IP
Layer 2
Layer 1
Upper Layer Message
Upper Layer Headers
Upper Layer (Application) Data
TCP/UDP Message
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
IP Datagram
IP Header
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
Layer 2 Frame
Layer 2 Header
IP Header
TCP/UDP Header
Upper Layer Headers
Upper Layer (Application) Data
Layer 2 Footer
1 0 0 1 0 1 1 0 1 1 0 0 1 0 1 1
Upper Layers
TCP / UDP
IP
Layer 2
Layer 1
The TCP/IP Guide


Application Layer
Transport Layer
Internet Layer
Link Layer
HTTP
process to process
HTTP
TCP
host to host
TCP
IP
IP
IP
IP
LAN
LAN
WAN
WAN
LAN
LAN

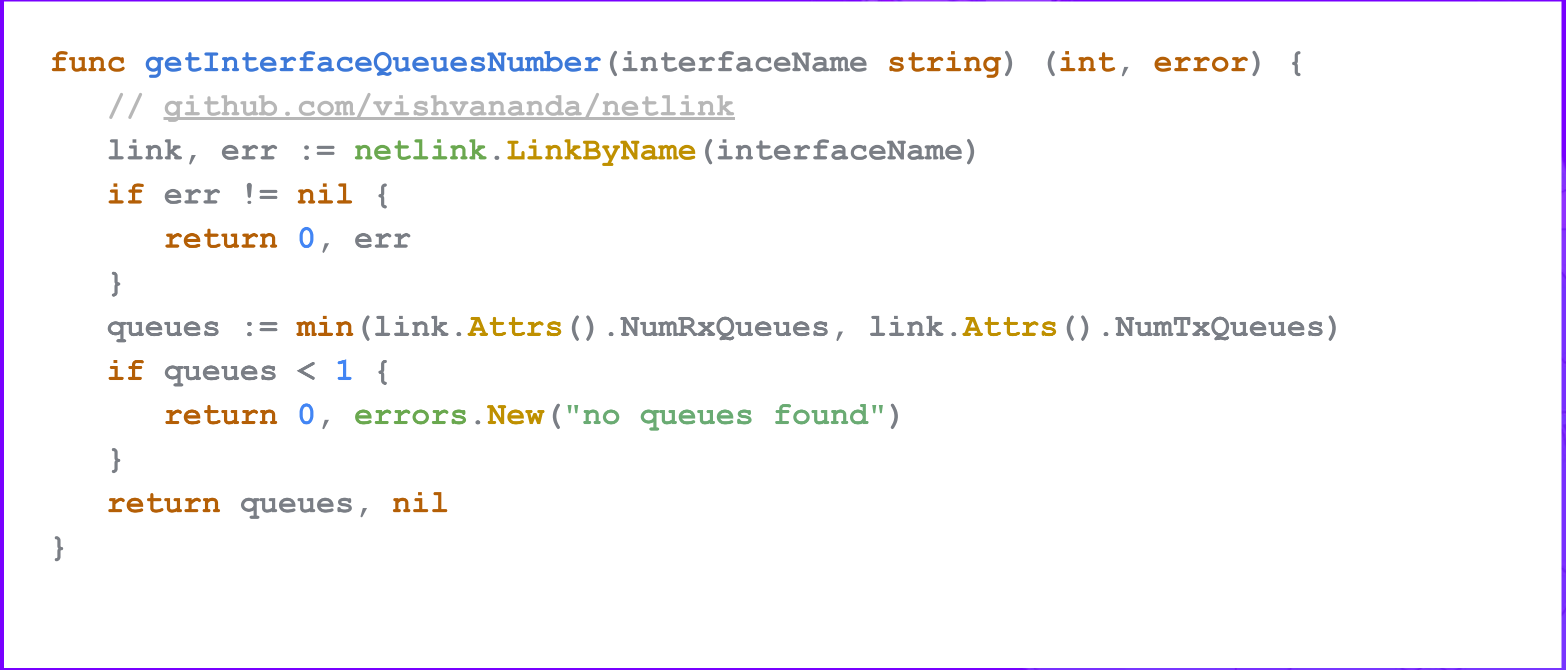
Network Interface Card (NIC)

Queues



Network

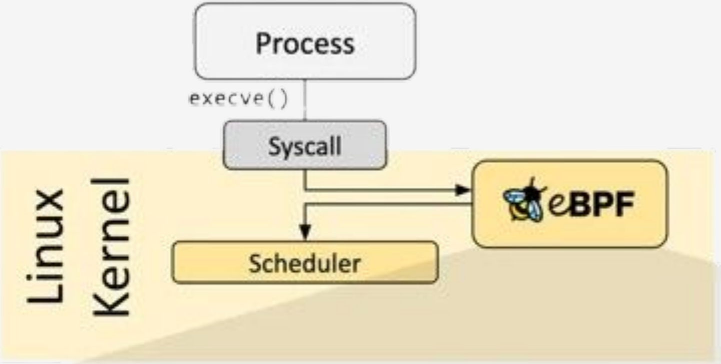
Queues



```go
func getInterfaceQueuesNumber(interfaceName string) (int, error) {
    // github.com/vishvananda/netlink
    link, err := netlink.LinkByName(interfaceName)
    if err != nil {
        return 0, err
    }
    queues := min(link.Attrs().NumRxQueues, link.Attrs().NumTxQueues)
    if queues < 1 {
        return 0, errors.New("no queues found")
    }
    return queues, nil
}
```

eBPF

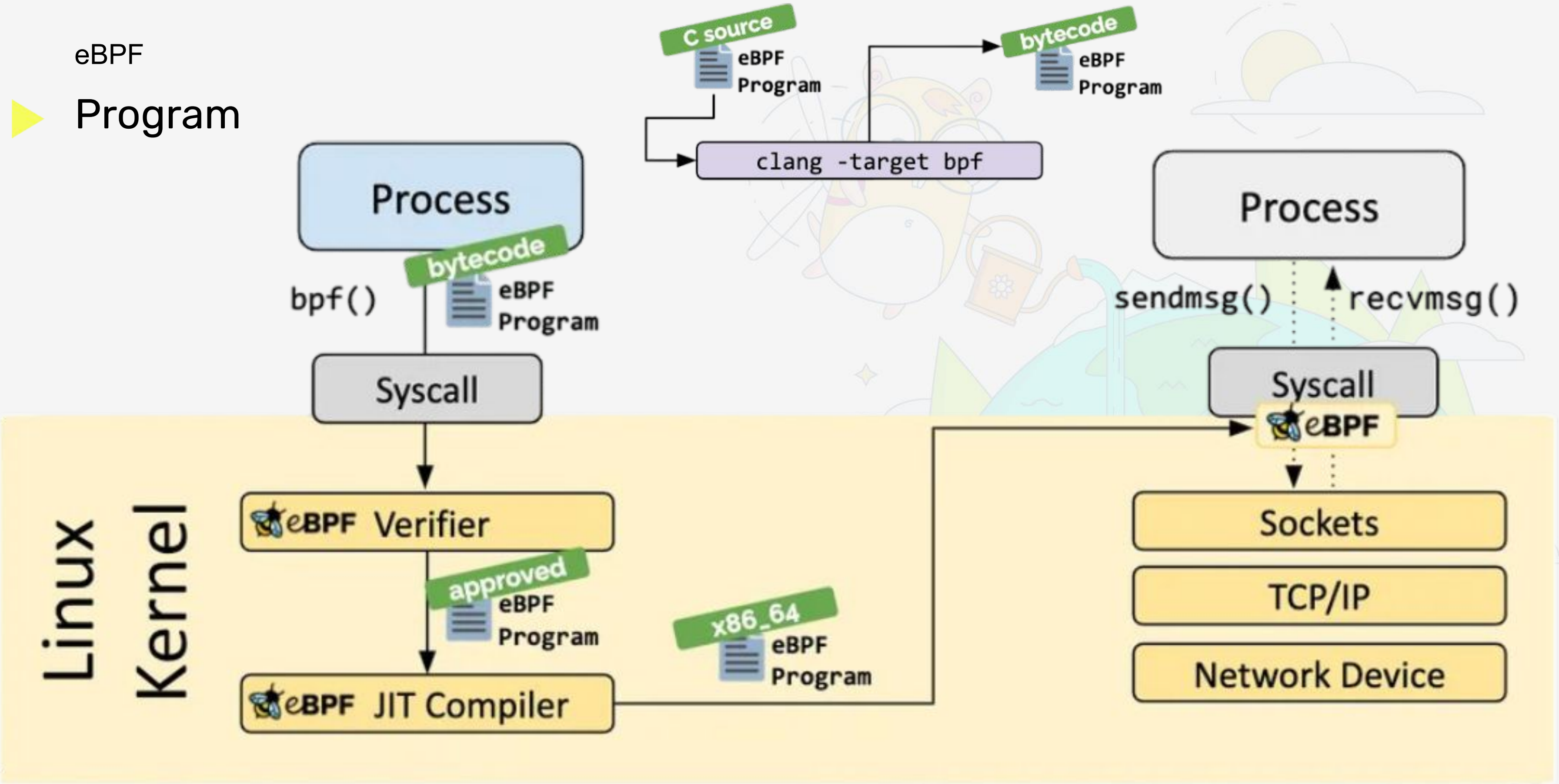
extended Berkeley Packet Filter



- Starting from Linux 3.18 (2014)
- Event-driven (system calls, network events, storage, etc.)
- eBPF program
  - Runs in a VM, within the kernel
  - Bytecode, JIT
  - Must be loaded with **bpf** system call (root permissions required by default)

eBPF

Program

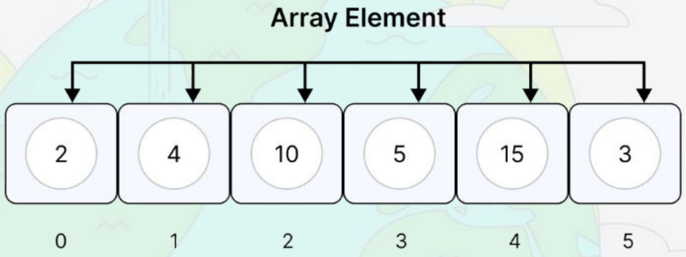


Maps

- They have a type assigned: e.g. BPF_MAP_TYPE_HASH (hashmap) or BPF_MAP_TYPE_ARRAY (array)

- Key-Value data structures

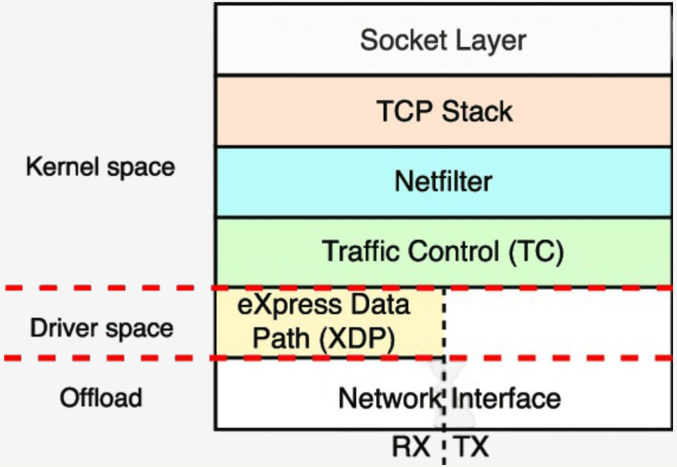
- Used to communicate with other eBPF programs or to receive data from the user space application



2	4	10	5	15	3
0	1	2	3	4	5

XDP

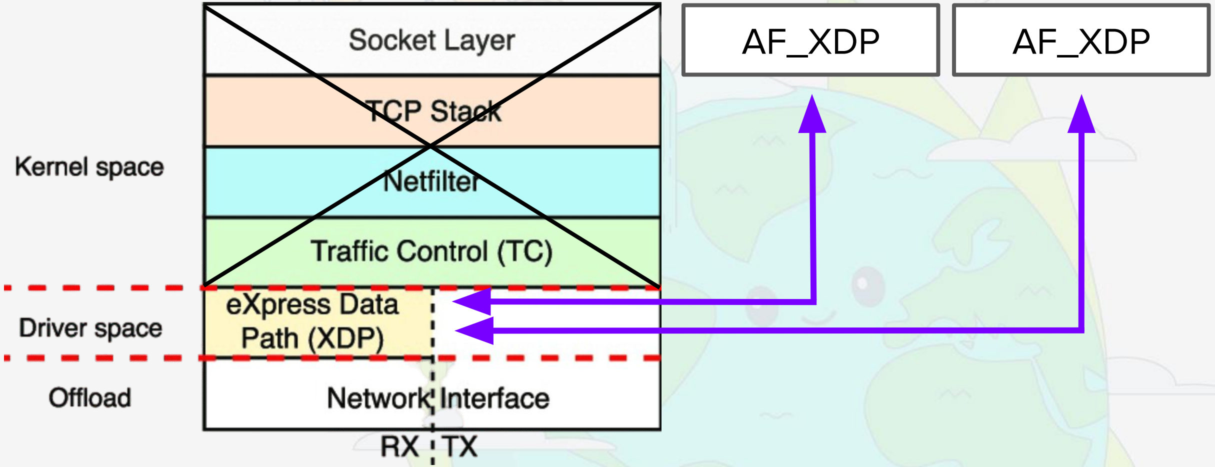
eXpress Data Path



- Starting from Linux 4.8 (2016)
- Built on top of eBPF
- eBPF program can run directly on the NIC (**offload**)
- XDP hook point is inside the NIC driver, called for **every incoming packet**
- When the NIC driver doesn't support XDP, eBPF program can be attached to the traditional net stack (but it's slower)

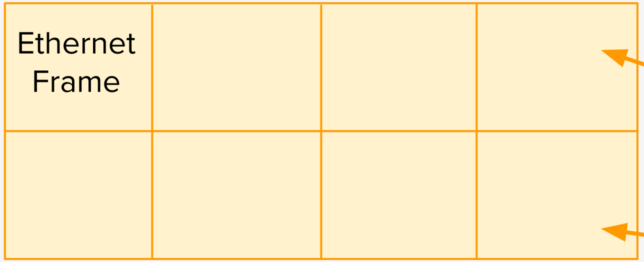
AF_XDP socket

- AF_XDP socket is created with the Linux socket() syscall
- Multiple sockets can process packets in parallel, each one assigned to a NIC queue
- The sockets can be passed to the eBPF program in a *BPF_MAP_TYPE_XSKMAP*

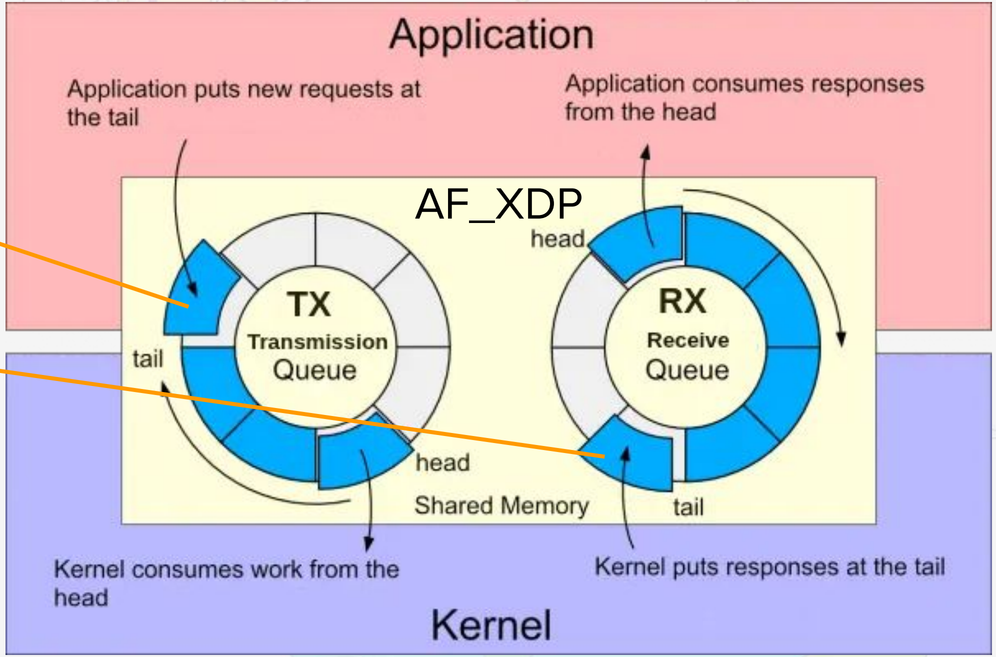


Memory model (simplified)

UMEM
(shared memory)



descriptor
*ptr + length


Application
Application puts new requests at the tail
Application consumes responses from the head
AF_XDP
head
TX
Transmission
Queue
RX
Receive
Queue
tail
head
Shared Memory
tail
Kernel consumes work from the head
Kernel puts responses at the tail
Kernel

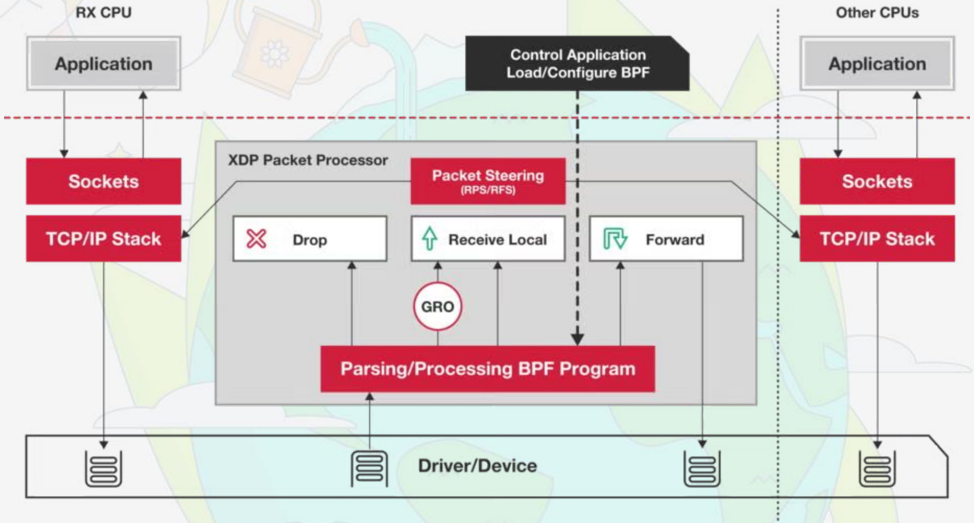
UMEM is allocated by user space application with a malloc.
It has 2 rings to handle its chunks ownership between kernel and userspace: FILL and COMPLETION rings.

XDP

Actions

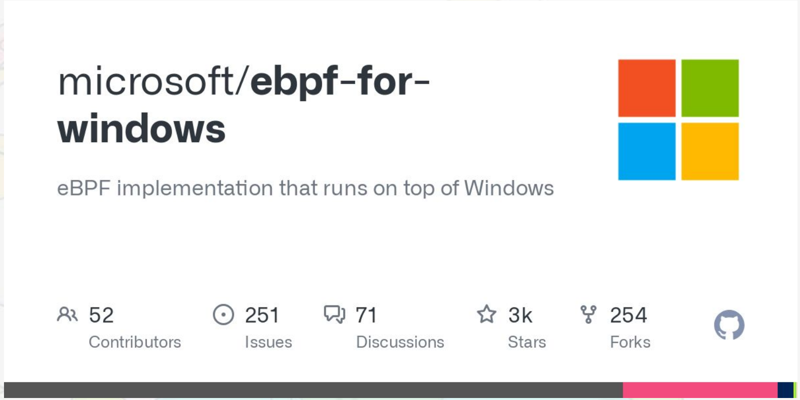
eBPF program return code:

- XDP_PASS (handled by OS net stack)
- XDP_DROP (dropped)
- XDP_TX (edited and retransmitted)
- XDP_ABORTED (eBPF program error code)
- XDP_REDIRECT (forwarded to another interface/socket)



eBPF & XDP for Windows

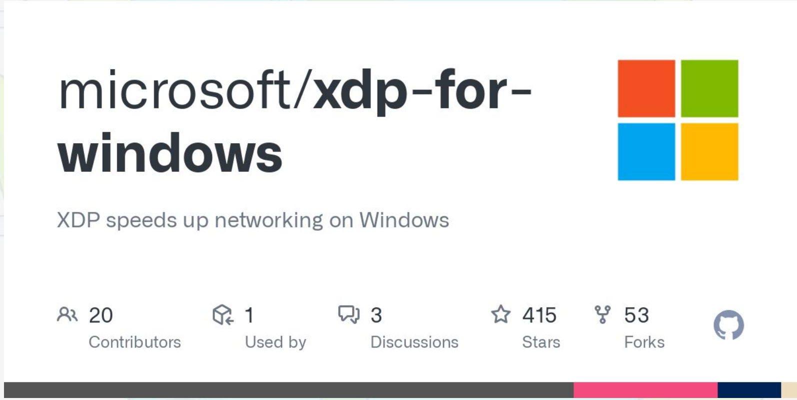
- Implementation is different compared to Linux
- Limited features
- Currently they aren’t part of Windows releases, they must be installed by the user



microsoft/**ebpf-for-windows**

eBPF implementation that runs on top of Windows

52 Contributors | 251 Issues | 71 Discussions | 3k Stars | 254 Forks



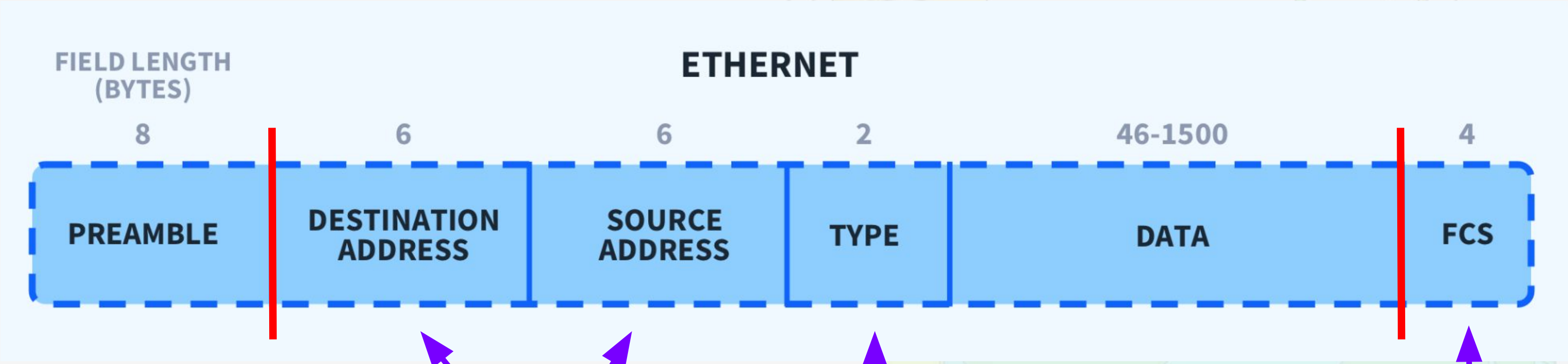
microsoft/**xdp-for-windows**

XDP speeds up networking on Windows

20 Contributors | 1 Used by | 3 Discussions | 415 Stars | 53 Forks

XDP

Ethernet



MAC Address
00:1A:2B:3C:4D:5E

0x0800: IPv4
0x86DD: IPv6

CRC
(checked
by NIC)

Ethernet

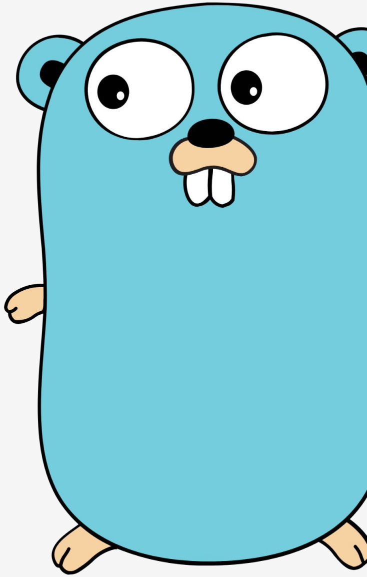
Parser & Serializer

```go
if len(receivedData) < 14 {
    return errors.New("invalid ethernet frame")
}
destinationAddress := receivedData[:6]
sourceAddress := receivedData[6:12]
etherType := binary.BigEndian.Uint16(
    receivedData[12:14],
)
payload := receivedData[14:]
```

```go
dataToSend := make(
    []byte, 14+len(payload),
)
copy(dataToSend[:6], destinationAddress)
copy(dataToSend[6:12], sourceAddress)
binary.BigEndian.PutUint16(
    dataToSend[12:14], uint16(0x0800),
)
copy(dataToSend[14:], payload)
```

“SouPPP” project

Domestic proxies project

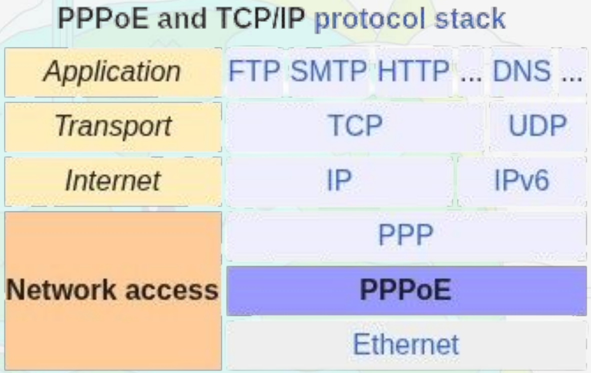


PPPoE Point-to-Point Protocol over Ethernet

Used by ISP to establish a connection with the home modem

Discovery (Ethernet Type 0x8863)
Initial handshake: Client sends an Ethernet broadcast message, Server sends a reply to the Ethernet source address.

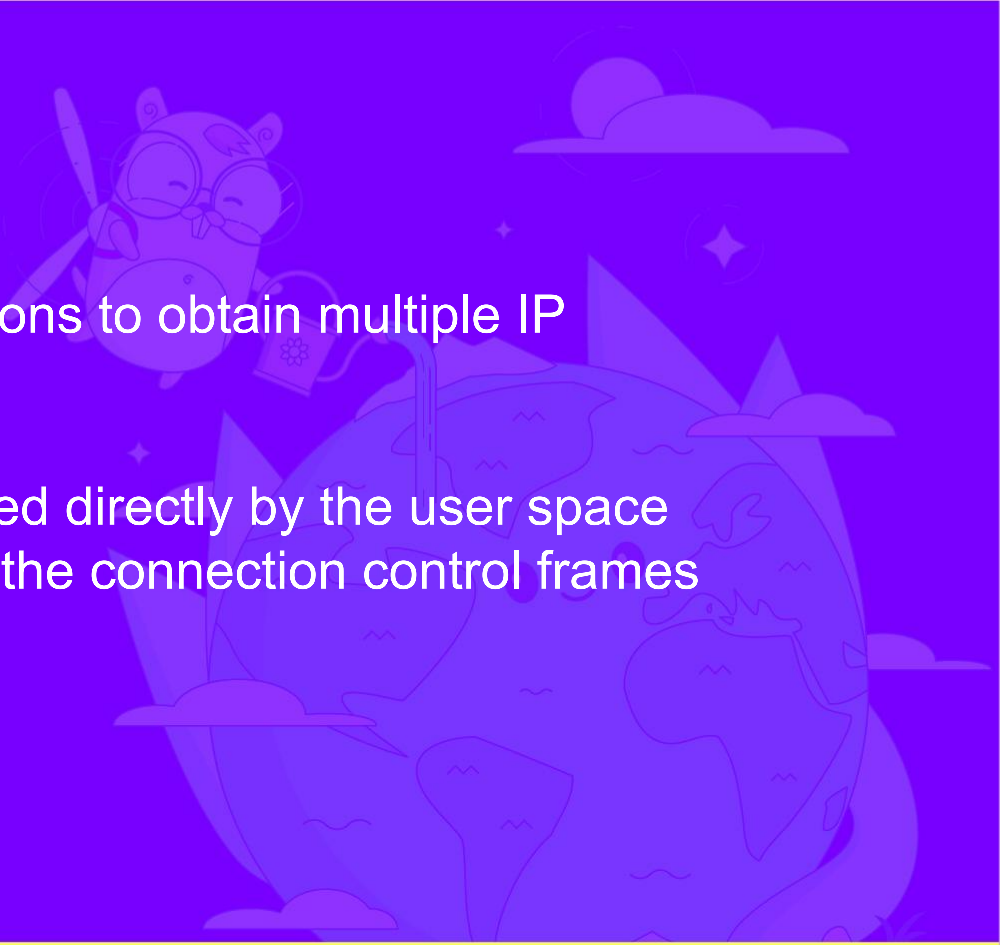
Session (Ethernet Type 0x8864)
Every IP packet is encapsulated in a PPP frame and sent/received over Ethernet with this ethernet type.



PPPoE and TCP/IP protocol stack

Application	FTP SMTP HTTP ... DNS ...	
Transport	TCP	UDP
Internet	IP	IPv6
Network access	PPP	
	PPPoE	
	Ethernet	

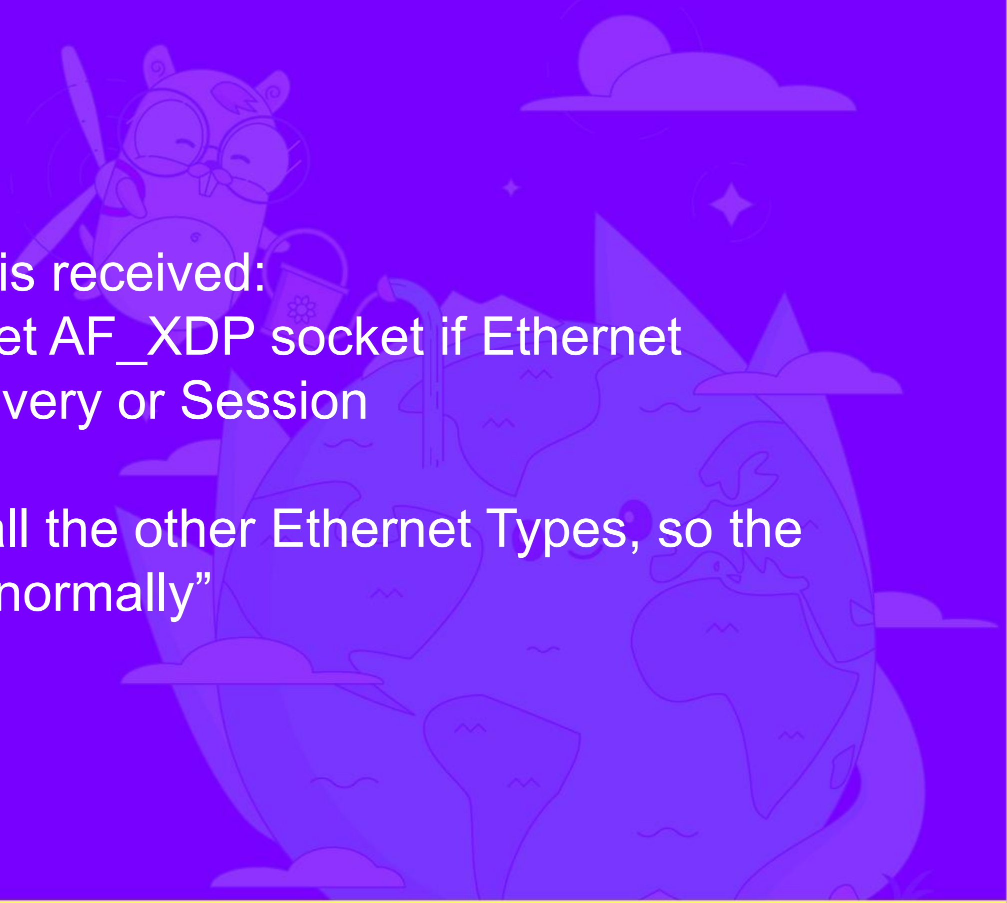
Features



- Multiple PPPoE sessions to obtain multiple IP addresses

- PPP session is handled directly by the user space application, including the connection control frames

XDP program (C, eBPF bytecode)



When an Ethernet frame is received:

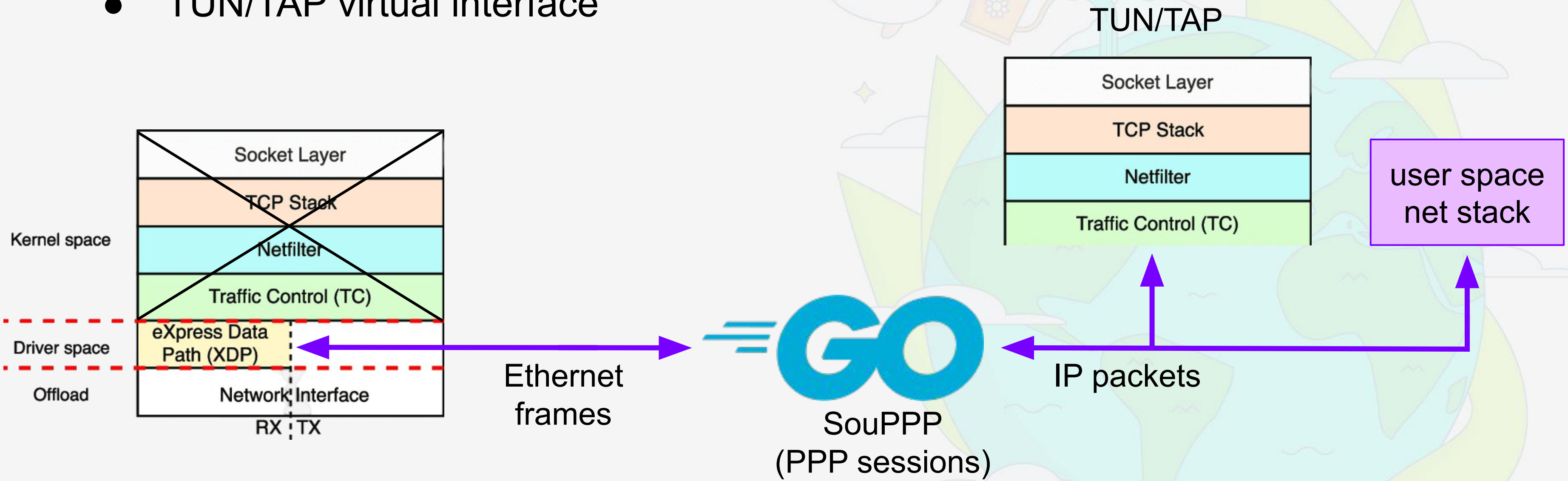
- Redirect it to the target AF_XDP socket if Ethernet Type is PPPoE Discovery or Session

- Use XDP_PASS for all the other Ethernet Types, so the kernel can handle it “normally”

Network traffic

Two alternatives:

- User space network stack (e.g. gVisor netstack)
- TUN/TAP virtual interface



Kernel space

Driver space

Offload

~~Socket Layer~~

~~TCP Stack~~

~~Netfilter~~

~~Traffic Control (TC)~~

eXpress Data Path (XDP)

Network Interface

RX TX

Ethernet frames

SouPPP (PPP sessions)

IP packets

TUN/TAP

Socket Layer

TCP Stack

Netfilter

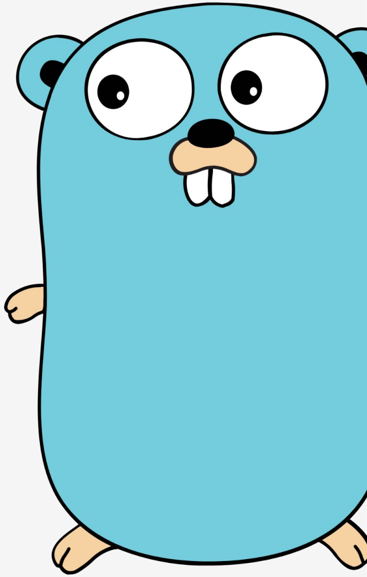
Traffic Control (TC)

user space net stack

HTTP requests

```go
dialer := &net.Dialer{ // We use it in &http.Transport{} passed to a &http.Client{}
  // Called after creating network connection, but before dialing
  ControlContext: func(ctx context.Context, network, address string, c
syscall.RawConn) error {
    var err error
    // Linux specific syscall to bind Dial() call to a specified network interface
    if ctrlErr := c.Control(func(fileDescriptor uintptr) {
        err = syscall.BindToDevice(int(fileDescriptor), "tunTapDeviceName")
    }); ctrlErr != nil {
        return ctrlErr
    }
    return err
  }
}
```

XDP in Go



Erik Pellizzon

erikpellizzon@gmail.com

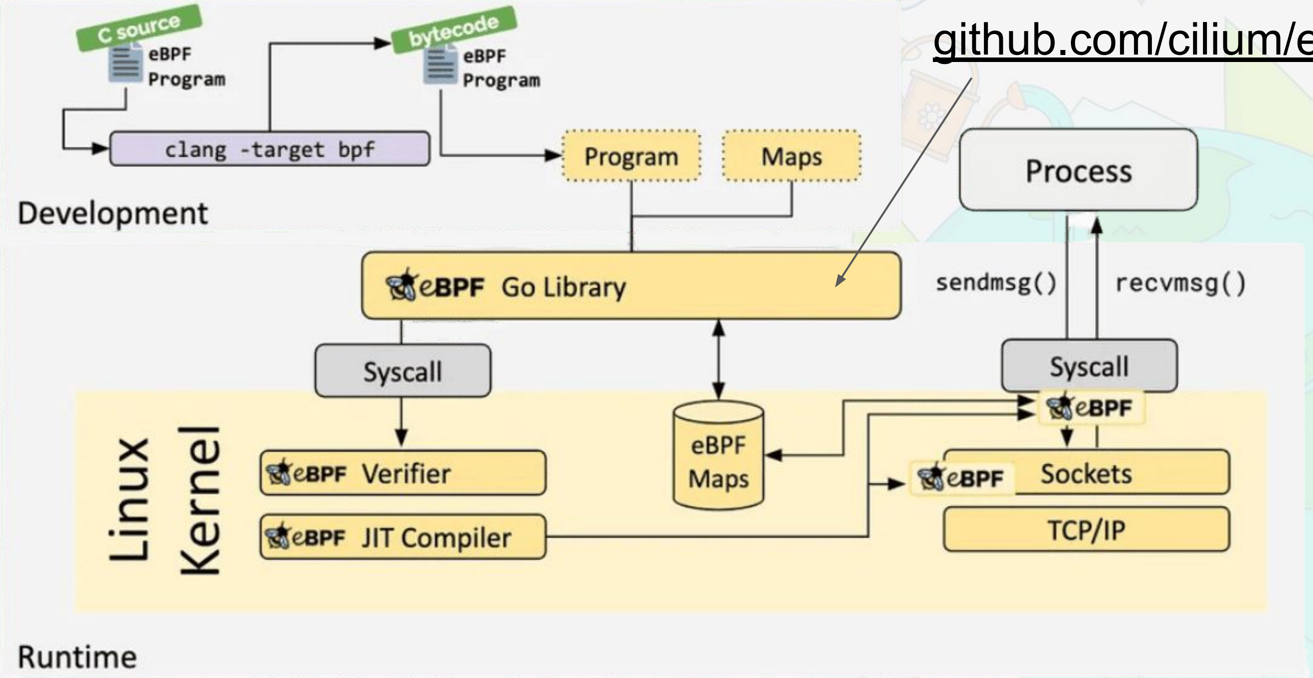
LinkedIn

Freelancer

Looking for collaborations

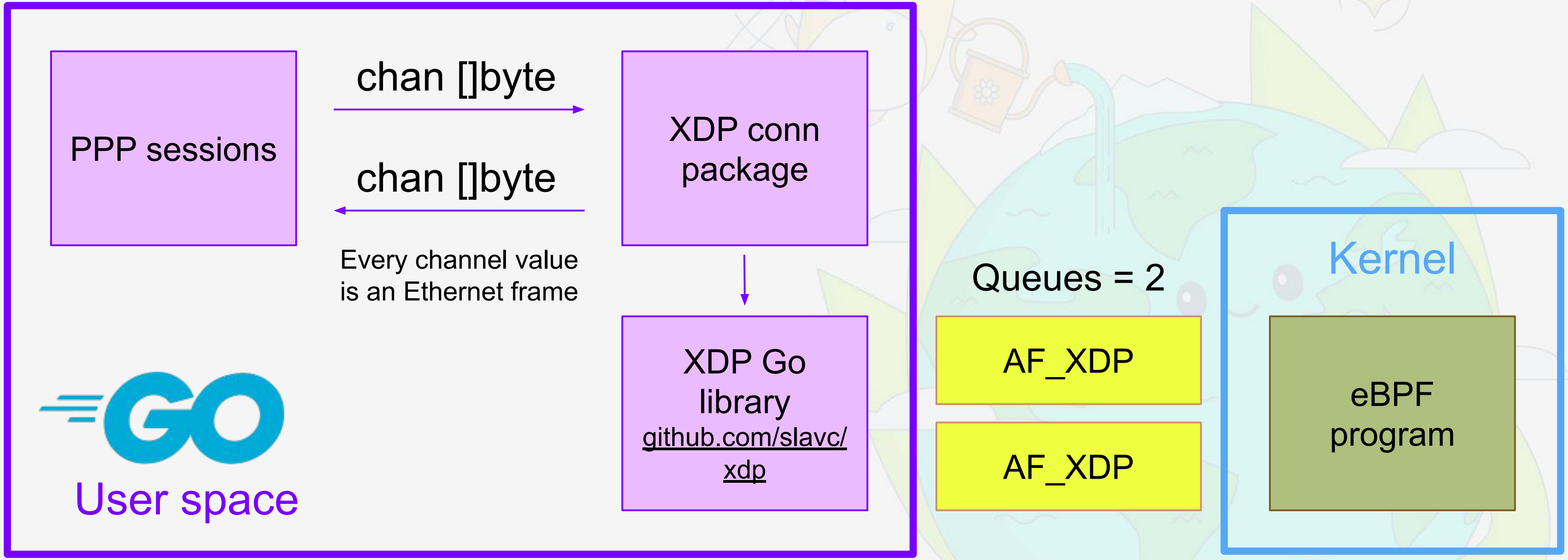
XDP in Go

eBPF program



github.com/cilium/ebpf

Code structure



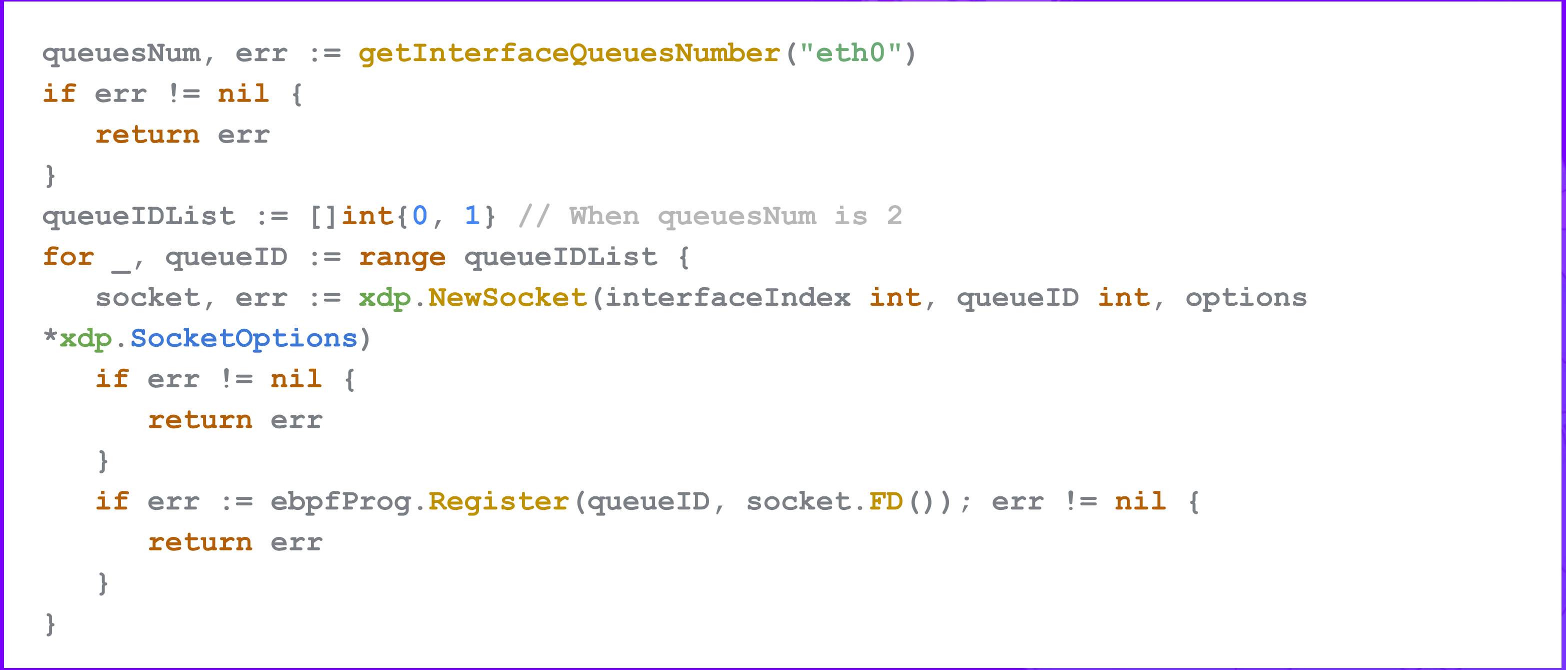
Initialization

```
interfaceData, err := netlink.LinkByName("eth0") // github.com/vishvananda/netlink
interfaceIndex := interfaceData.Attrs().Index

ebpfProg, err := func() (*xdp.Program, error) {
   spec, err := ebpf.LoadCollectionSpecFromReader(ebpfBytecodeReader)
   col, err := ebpf.NewCollection(spec)
   err = col.Maps["ethernetTypeInputMap"].Put(0x8863, 1) // PPPoE Discovery
   err = col.Maps["ethernetTypeInputMap"].Put(0x8864, 1) // PPPoE Session
   return &xdp.Program{Program: col.Programs[…], Queues: col.Maps[…], Sockets:
col.Maps[…]}, nil
}()

// Attach the XDP Program to the network interface
err = ebpfProg.Attach(interfaceIndex)
```

Sockets



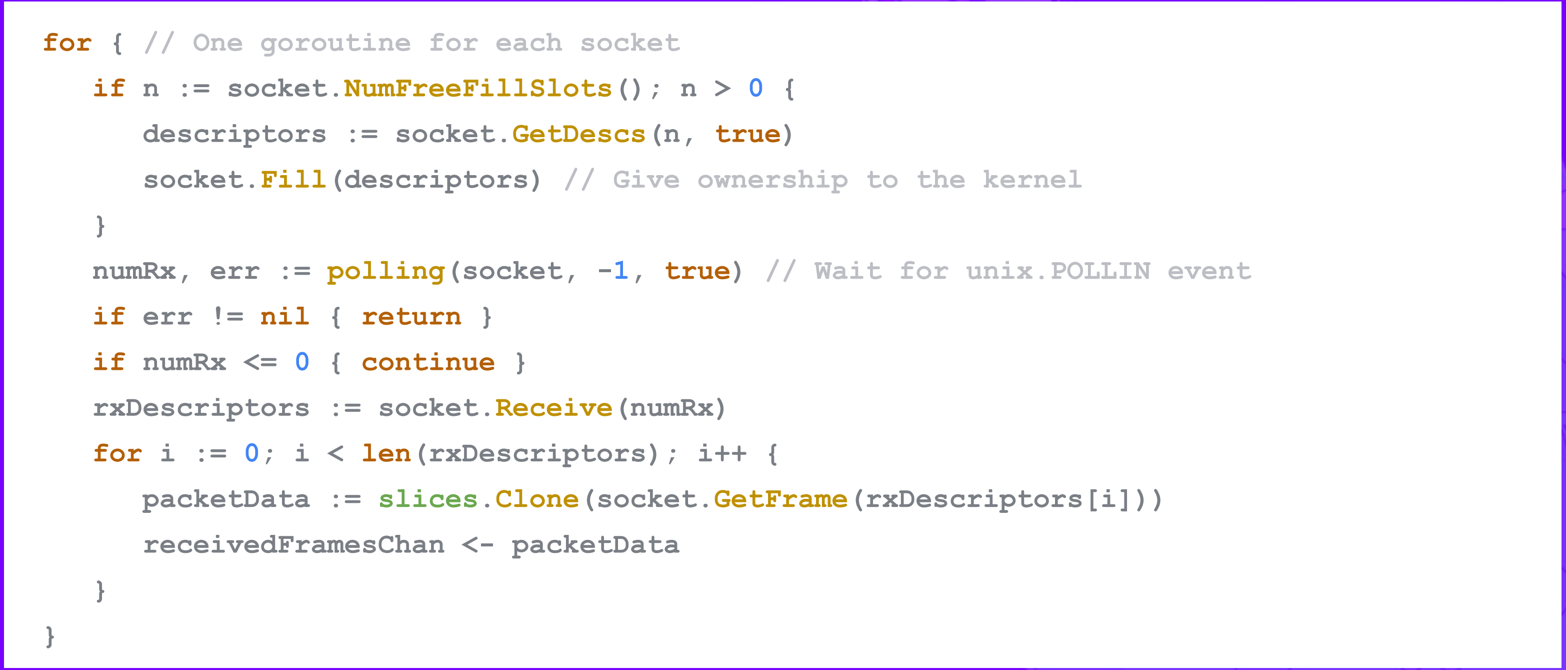
```go
queuesNum, err := getInterfaceQueuesNumber("eth0")
if err != nil {
   return err
}
queueIDList := []int{0, 1} // When queuesNum is 2
for _, queueID := range queueIDList {
   socket, err := xdp.NewSocket(interfaceIndex int, queueID int, options
*xdp.SocketOptions)
   if err != nil {
      return err
   }
   if err := ebpfProg.Register(queueID, socket.FD()); err != nil {
      return err
   }
}
```

XDP in Go

Transmission

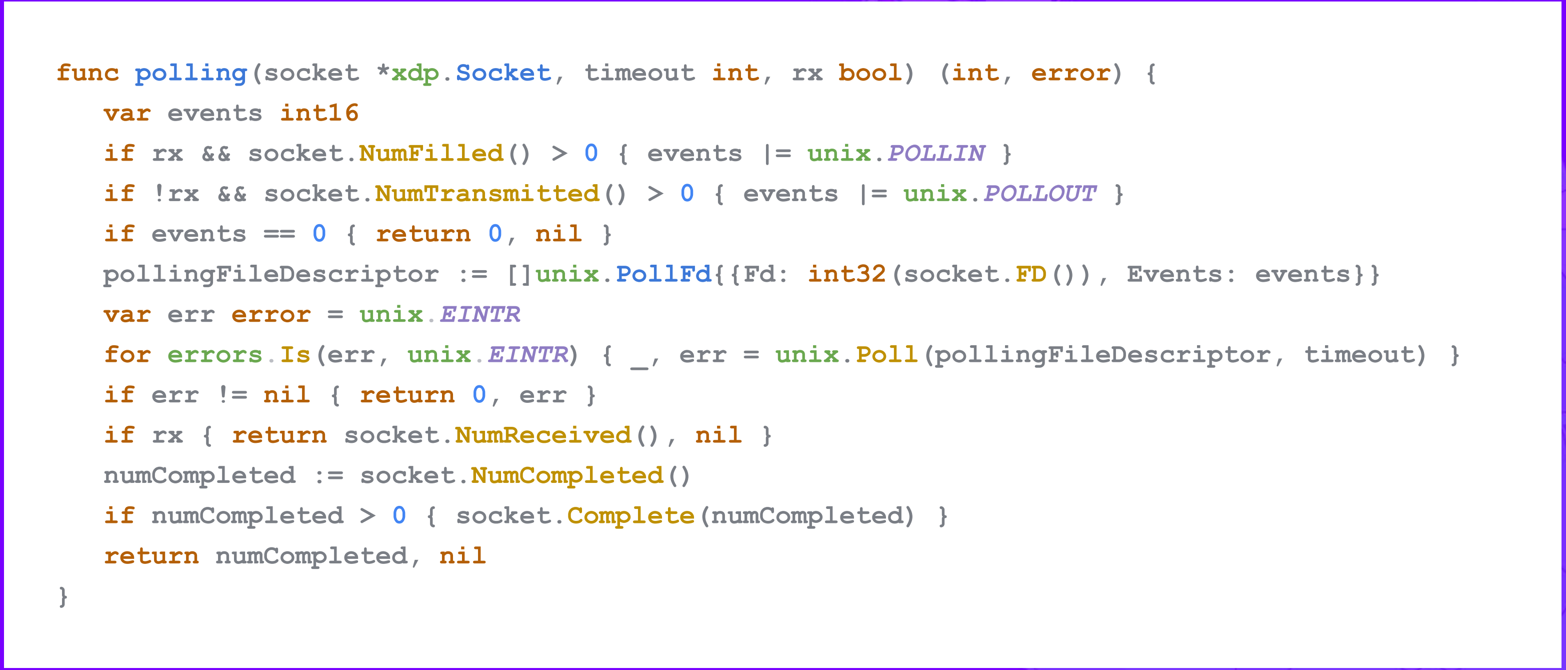
```go
for { // One goroutine for each socket
    data := <-framesToSendChan; numFrames := 1
    descriptors := socket.GetDescs(numFrames, false) // n int, rx bool
    if len(descriptors) < numFrames { return }
    // GetFrame() returns the underlying []byte buffer of the descriptor
    copy(socket.GetFrame(descriptors[0]), data)
    descriptors[0].Len = uint32(len(data))
    numSubmitted := socket.Transmit(descriptors)
    if numSubmitted != numFrames { return }
    numTx, err := polling(socket, -1, false) // Wait for unix.POLLOUT event
    if err != nil { return }
}
```

Receive



```go
for { // One goroutine for each socket
   if n := socket.NumFreeFillSlots(); n > 0 {
      descriptors := socket.GetDescs(n, true)
      socket.Fill(descriptors) // Give ownership to the kernel
   }
   numRx, err := polling(socket, -1, true) // Wait for unix.POLLIN event
   if err != nil { return }
   if numRx <= 0 { continue }
   rxDescriptors := socket.Receive(numRx)
   for i := 0; i < len(rxDescriptors); i++ {
      packetData := slices.Clone(socket.GetFrame(rxDescriptors[i]))
      receivedFramesChan <- packetData
   }
}
```

Polling



```go
func polling(socket *xdp.Socket, timeout int, rx bool) (int, error) {
    var events int16
    if rx && socket.NumFilled() > 0 { events |= unix.POLLIN }
    if !rx && socket.NumTransmitted() > 0 { events |= unix.POLLOUT }
    if events == 0 { return 0, nil }
    pollingFileDescriptor := []unix.PollFd{{Fd: int32(socket.FD()), Events: events}}
    var err error = unix.EINTR
    for errors.Is(err, unix.EINTR) { _, err = unix.Poll(pollingFileDescriptor, timeout) }
    if err != nil { return 0, err }
    if rx { return socket.NumReceived(), nil }
    numCompleted := socket.NumCompleted()
    if numCompleted > 0 { socket.Complete(numCompleted) }
    return numCompleted, nil
}
```

Greetings and conclusion

Erik Pellizzon

erikpellizzon@gmail.com

LinkedIn

Freelancer

Looking for collaborations