# Packet Processing Frameworks

## Basics, BPF and PFil

George Neville-Neil

Neville-Neil Consulting

AsiaBSDCon March 2014

# Intro

- Overview
- Berkeley Packet Filter (BPF)
- pfil
- IPFW and Dummynet
- PF in FreeBSD
- Netmap
- Netgraph

# Speaker Bio

- Twenty year veteran of BSDs and Operating Systems in general
- Coauthor of *The Design and Implementation of the FreeBSD Operating System*
- Former member of FreeBSD's Elected Core Team
- Director of the FreeBSD Foundation
- Network Protocol and Security Practitioner
- ACM Queue's Kode Vicious

# Why do they exist?

- Packet filtering
- Debugging
- Kernel bypass
- Access to full packet headers
- New protocol development

# Class Overview

- Necessary Data Structures
- BPF
- PFIL
- IPFW and Dummynet
- Netmap
- Netgraph

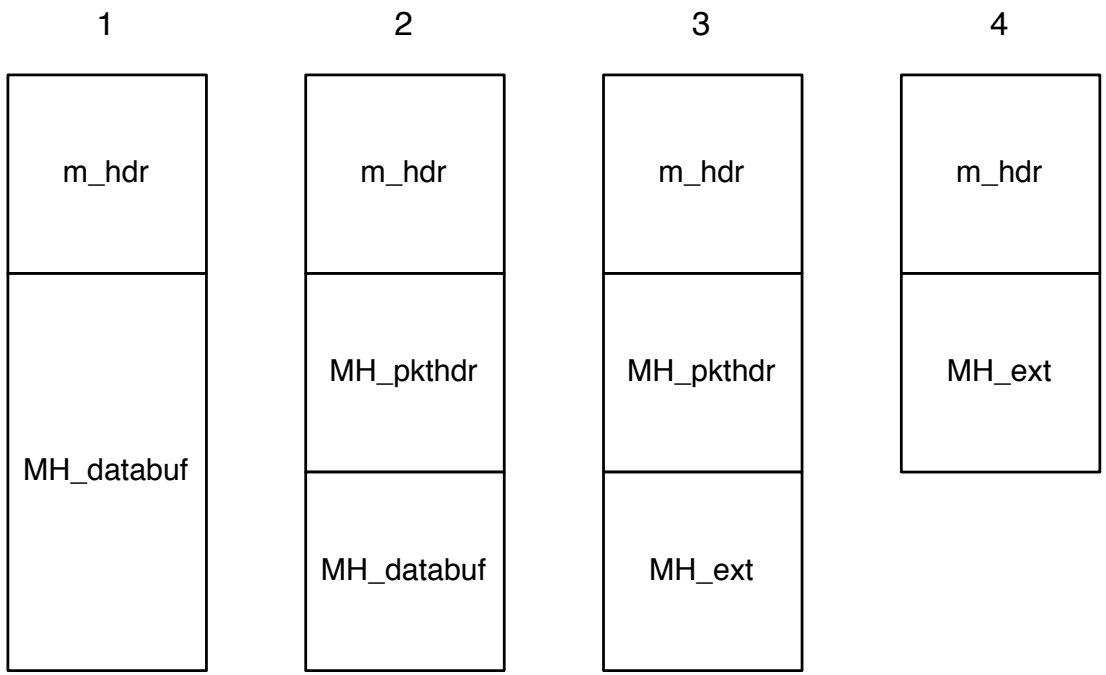
# Memory for Packets

- Packets need to be stored for reception and transmission
- The basic packet memory structures are the `mbuf` and `cluster`
- `mbuf` structures have several types and purposes
- Clusters hold only data
- History dictates that `mbuf`s are named `m`
- In the kernel we will see many pointers to `mbuf`s

# Types of mbufs

- Wholly contained
- Packet Header
- Using a cluster

# Four Types of mbufs



# Code for an mbuf structure

```
struct mbuf {
        struct m_hdr     m_hdr;
        union {
                struct {
                        struct pkthdr    MH_pkthdr;       /* M_PKTHDR set */
                        union {
                                struct m_ext     MH_ext; /* M_EXT set */
                                char             MH_databuf[MHLEN];
                        } MH_dat;
                } MH;
                char     M_databuf[MLEN];                 /* !M_PKTHDR, !M_EXT */
        } M_dat;
};
```

# Handling mbufs

- Limits
- Allocation types
- Reference counting

# mbuf Code Walk Through

# Virtual Networking (VNET)

- Multiple instances of networking code
- Identified by `struct vnet`
- All new network subsystems must be virtualized
- Will not be covered in detail in this class

# Break

# BPF

- Earliest Packet Framework
- Debugging
- tcpdump
- libpcap

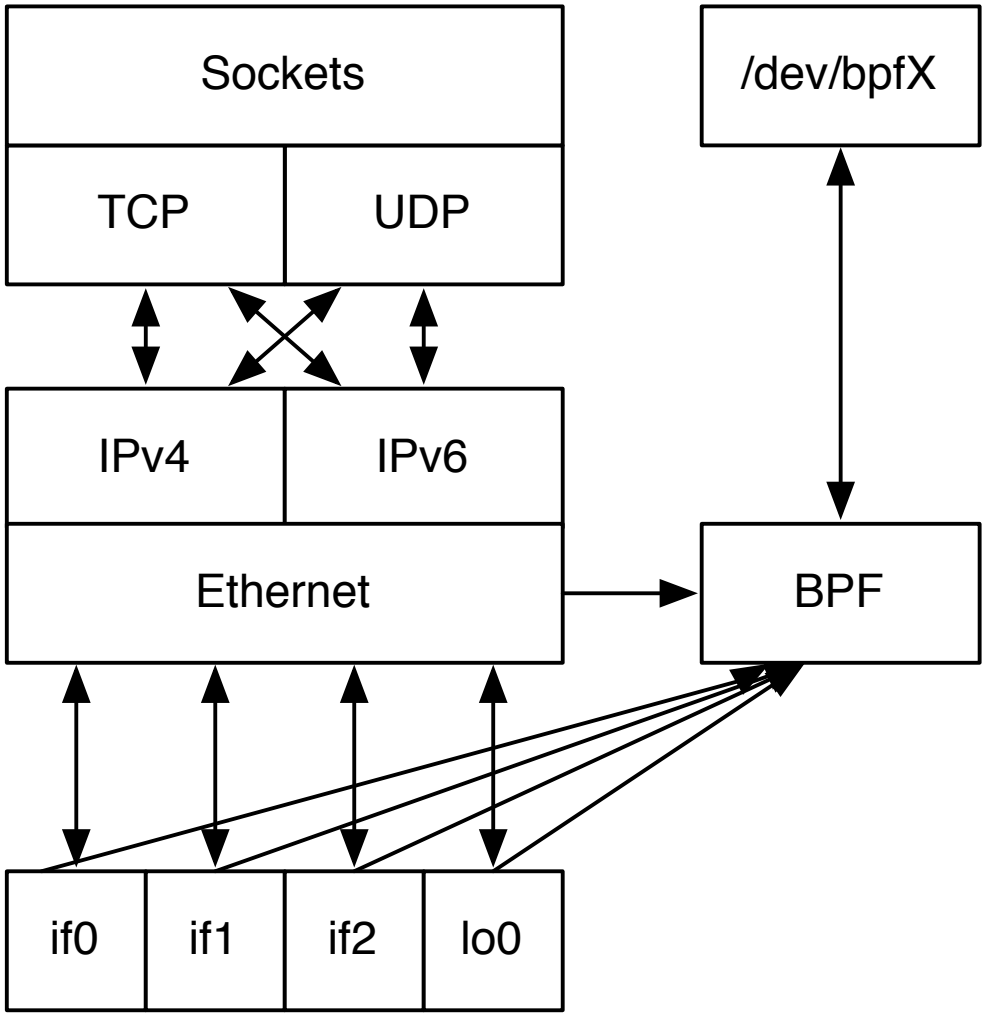
# Uses

- Debugging
- Low Jitter Protocols
- Cannot be used as a firewall

# Theory of Operation

- Capture Packet
- Examine Contents
- Match Listeners
- Copy Packet
- Continue normal processing

# Location in the Stack



# The BPF Device

- Pseudo device (/dev/bpf0)
- Enabled by default in all kernels
- Supports open/close/read/write/ioctl interface
- Programmatically controlled via ioctl()

# BPF ioctl interface

- BIOCSETF
- BIOCFLUSH
- BIOCPROMISC
- BIOCGDLTLIST
- BIOCGDLT
- BIOC[GET|SET]IF
- BIOC[G|S]RTIMEOUT
- BIOCGSTATS
- BIOCIMMEDIATE
- BIOCVERSION
- BIOC[G|S]RSIG
- BIOC[G|S]HDRCMPLT
- BIOC[G|S]DIRECTION
- BIOCLOCK
- BIOCSETWF
- BIOCFEEDBACK
- BIOCGETBUFMODE
- BIOCSETBUFMODE
- BIOCGETZMAX
- BIOCROTZBUF
- BIOCSETZBUF
- BIOCSETFNR
- BIOC[G|S]TSTAMP

# How does it work?

- Driver Hooks
- BPF Programs
- Libpcap simplifies access

# Driver Hooks

- Network Drivers must support BPF for output
- Ethernet input routines handle inbound packets
- `BPF_TAP`, `BPF_MTAP`, `BPF_MTAP2`
- `catchpacket()`

# Driver Capture Points

```
                BUS_DMASYNC_POSTWRITE);
2               sc->tx_descs[sc->txhead].addr = segs[0].ds_addr;
```

- Example from the ATE driver if_ate.c
- Add drivers effectively the same
- Called with interface pointer and mbuf

# BPF Programs

- Assembly Language for Packets
- Difficult for non programmers
- Optimizable
- Re targetable to real hardware

# BPF Example

```
struct bpf_insn insns[] = {
    BPF_STMT(BPF_LD+BPF_H+BPF_ABS, 12),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, ETHERTYPE_IP, 0, 8),
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 26),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 2),
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 3, 4),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x80037023, 0, 3),
    BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 30),
    BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8003700f, 0, 1),
    BPF_STMT(BPF_RET+BPF_K, (u_int)-1),
    BPF_STMT(BPF_RET+BPF_K, 0),
};
```

# What does this do?

# What does this do?

- Accepts only IP packets between host 128.3.112.15 and 128.3.112.35

# Libpcap and BPF

- Libpcap simplifies access
- Easier to understand syntax
- proto ip and host 128.3.112.15 or host 128.3.112.35

# BPF Code Walk Through

# Break

- Please return in 15 minutes

# pfil

- generic packet filtering points throughout the kernel
- used by all other packet processing frameworks

# PFil Consumers

- ipfw
- PF
- ipfilter
- SIFTR

# Current Locations

- Bridge
- ENC Device
- ether_output/ether_demux
- ip_input/ip_output
- ip6_input/ip6_output

# Theory of Operation

- Capture packets at various points
- Run packets through one or more hooks
- Return value indicates disposition of the packet
  - 0 to continue
  - errno to stop processing the packet
- Filtering function is responsible for mbuf cleanup
- Hooks can have multiple filters

# Hook Prototype

```
int (*pfil_func_t)(
        void *,                 /* Pointer to arbitrary data */
         struct mbuf **,    /* Packet data, can be replaced */
         struct ifnet *,        /* Interface from which the packet originated */
         int,                       /* Direction */
         struct inpcb *    /* Protocol Control Block */
         );
```

# Adding a Hook

```
pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
if (pfh_inet == NULL)
        return (ESRCH); /* XXX */
pfil_add_hook(pf_check_in, NULL, PFIL_IN | PFIL_WAITOK, pfh_inet);
```

# Example Hook

```
pf_check_in(void *arg, struct mbuf **m, struct ifnet *ifp, int dir,
    struct inpcb *inp)
{
        int chk;

        chk = pf_test(PF_IN, ifp, m, inp);
        if (chk && *m) {
            m_freem(*m);     /* We are responsible for freeing state */
            *m = NULL;
        }

        return (chk);
}
```

# PFIL Code Walk Through

# Summary of Part 1

- Not all networking code is TCP/IP
- BPF is the lowest layer packet filter
- PFil is the basis of all other packet processing frameworks

# Coming in Part 2

- IPFW and Dummynet
- PF in FreeBSD
- Netmap
- Netgraph

# Break

# Review of Material from Part 1

- Overview
- Berkeley Packet Filter (BPF)
- pfil

# Overview Part 2

- IPFW and Dummynet
- PF in FreeBSD
- Netmap
- Netgraph

# IPFW and Dummynet

- FreeBSD Firewall
- Does more than provide a firewall
  - Delays
  - Queueing
  - Bandwidth shaping
- builds on on concepts from BPF

# Theory of Operation

- A single pfil hook (ipfw_check_hook)
  - ip_input
  - ip6_output
  - ip6_forward
  - ip_input
  - ip_output

# IPFW capture points

- Link Layer
- IPv4
- IPv6

# Filter Matching

- Rules are expressed as instructions
- Very similar to BPF
- Some added instructions specific to ipfw
  - PIPE
  - QUEUE
  - DIVERT
  - TEE

# Rule Lookup

- Centralized via ipfw_chk routine
- Called from

  ipfw_check_frame Link layer filters
  ipfw_check_packet IPv4 and IPv6 filters
- Explained further during code walk through

# Disposition

IP_FW_PASS Accept the packet

IP_FW_DENY Drop the packet

IP_FW_DIVERT Divert packet to another port

IP_FW_TEE Copy the packet packet, pass and send to another port

IP_FW_DUMMYNET Send to Dummynet pipe

IP_FW_NETGRAPH Hand to netgraph

# IPFW Code Walk Through

# Dummynet

- Bandwidth Shaping
- Delay and Loss Emulation
- Integrated with ipfw
- Initially developed for testing TCP congestion control

# Dummynet Structure

- Packets are tagged
- Assigned to a flowset
- Scheduled through the system

# Pipes, Queues, Delaying and Drops

- Handled via centralized code
- Each scheduler is its own file
  - FIFO
  - Priority
  - Round Robin
  - Weighted Fair Queueing
  - QFQ, a variant of Weighted Fair Queueing

# Dummynet Walk Through

# Break

- Please return in 15 minutes

# PF

- Originally imported from OpenBSD
- Most popular packet filter on the BSDs
  - Because of its rule language
- Uses pfil hooks

# Matching a Packet

- pf_test routine starts lookup
- pf_test6 used for IPv6
- Higher layer protocols have their own tests
  - pf_test_state_icmp
  - pf_test_state_udp
  - pf_test_state_tcp
  - pf_test_state_other
- No generalized filtering system

# Packet Disposition

| | |
|---|---|
| PF_PASS | Allow packet to proceed |
| PF_DROP | Drop packet |
| PF_SCRUB | Scrub TCP flags |
| PF_NAT | Network Address Translation |
| PF_BINAT | Bidirectional NAT Mapping |
| PF_RDR | Redirection |
| PF_SYNPROXY_DROP | Perform 3-way handshake |
| PF_DEFER | Part of pfsync |

# PF Walk Through

# Break

- Please return in 15 minutes

# Netmap

- Kernel bypass
- Exposes NIC queues in user space
- Direct Access to packets

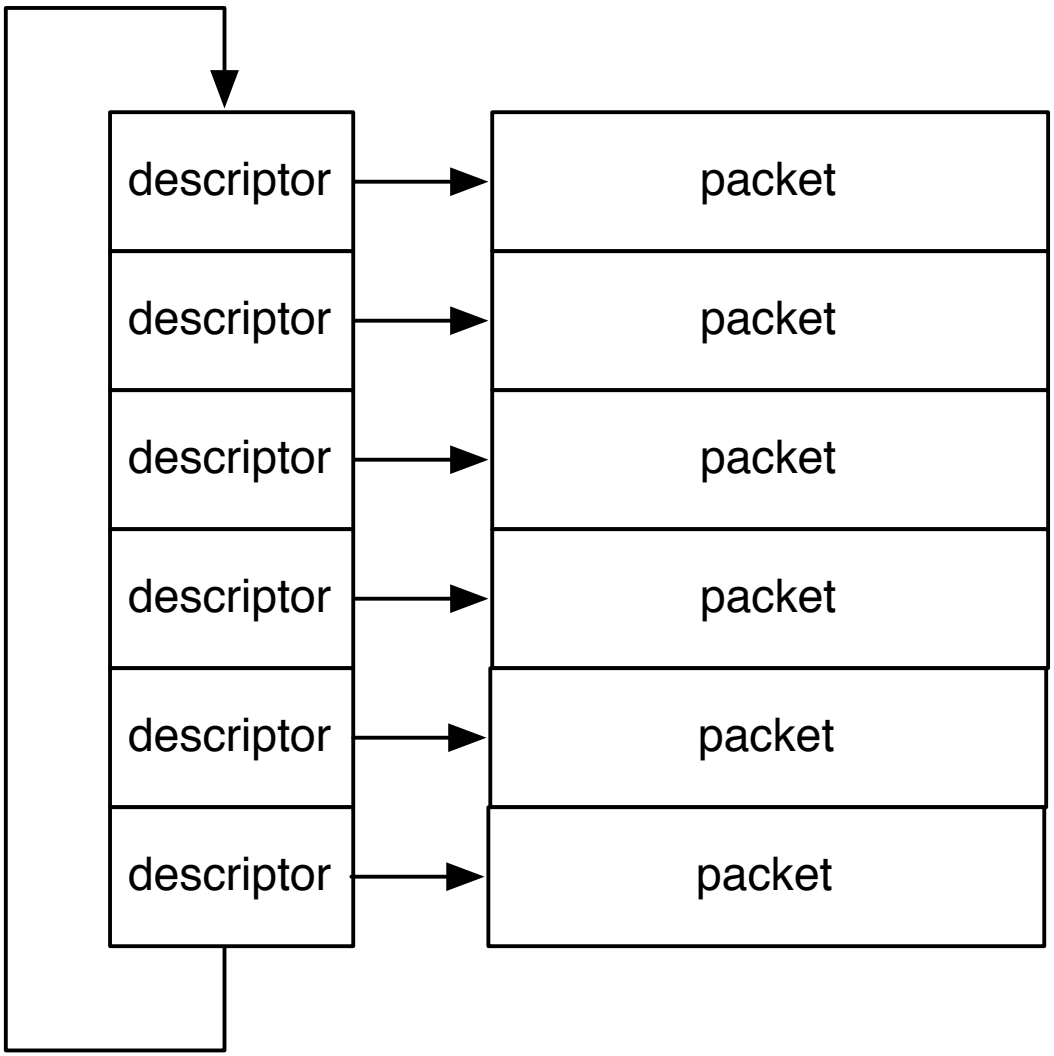
# Theory of Operation

- Direct access to hardware
- Maps device structures into user space
- Depends on kernel for safe synchronization

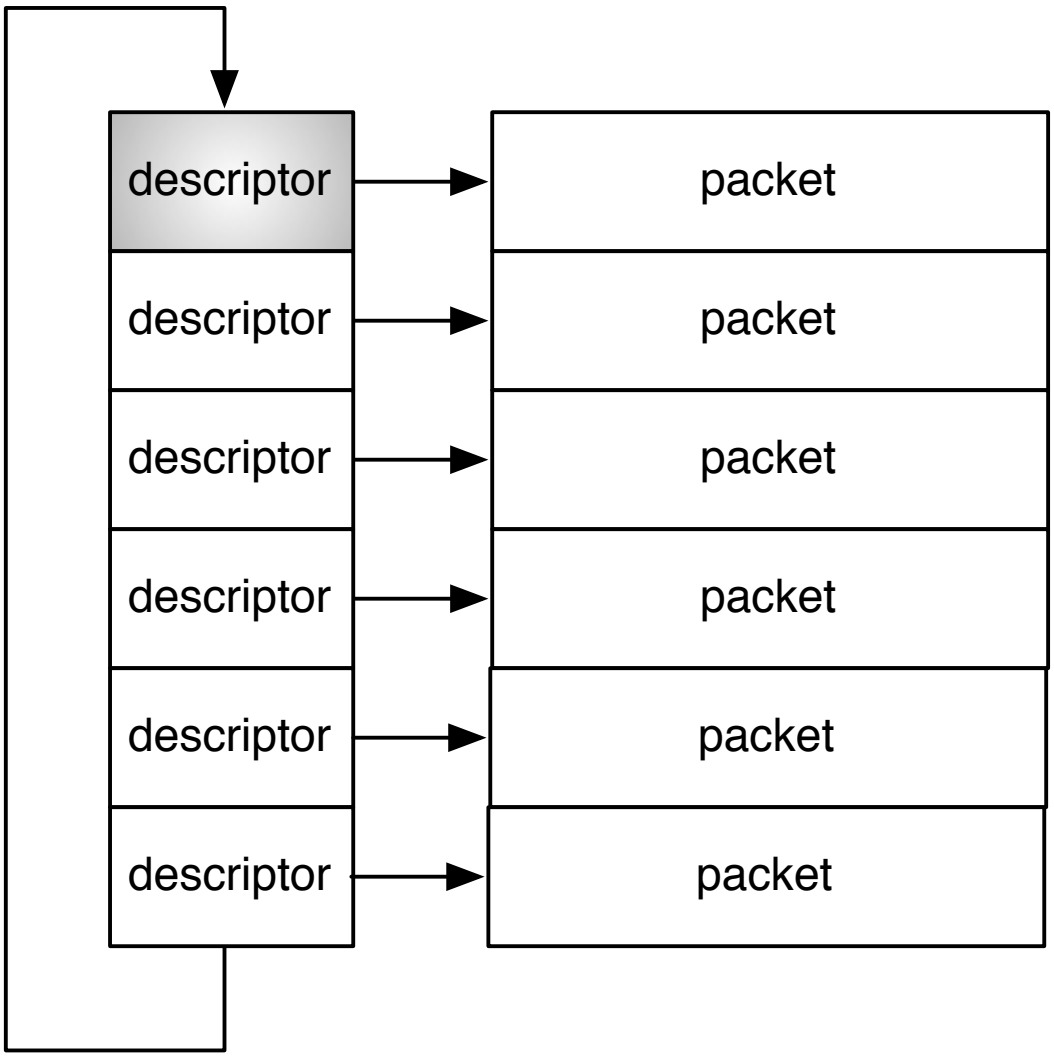
# How does a NIC work?

- Rings of Packet Descriptors
- Synchronization Bit
- DMA Engine
- Head and Tail Pointers

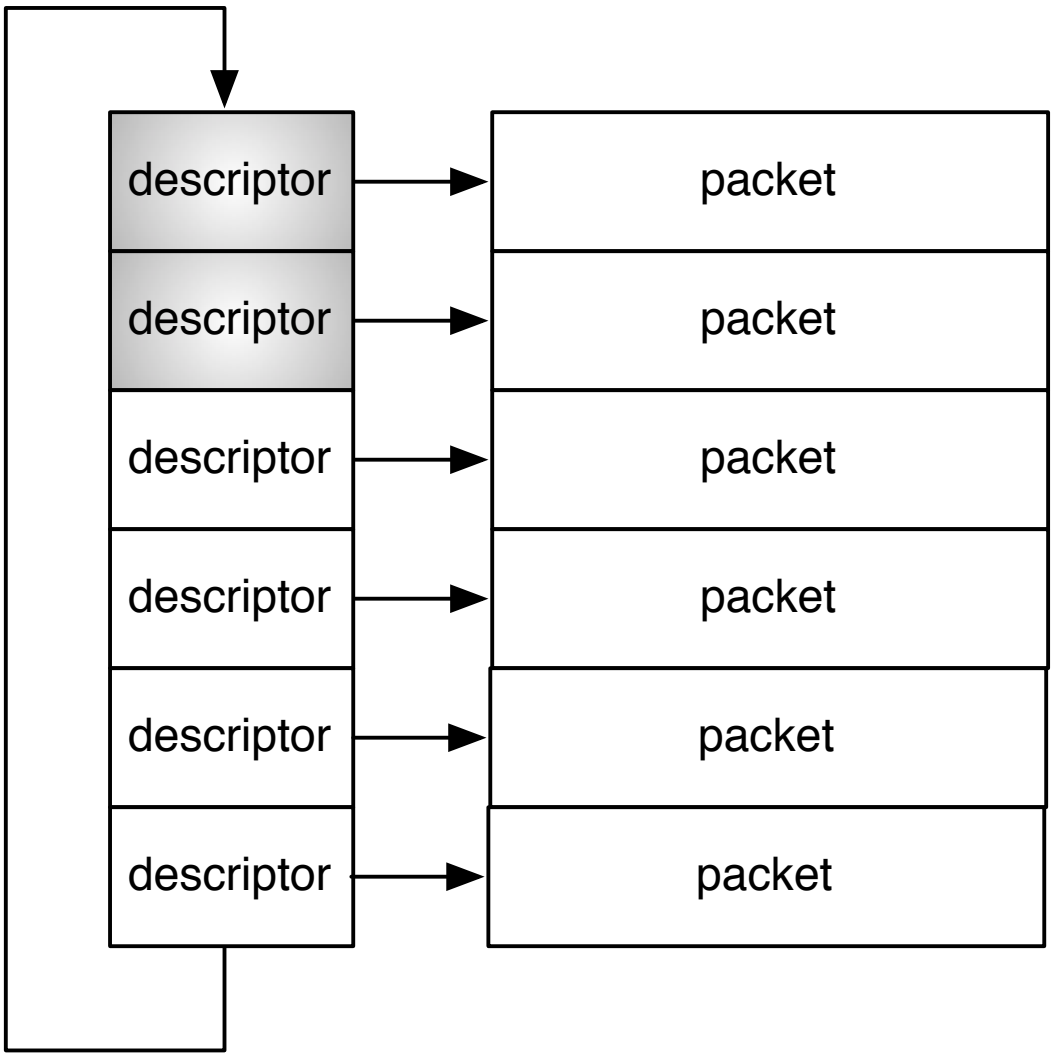
# Receive Ring



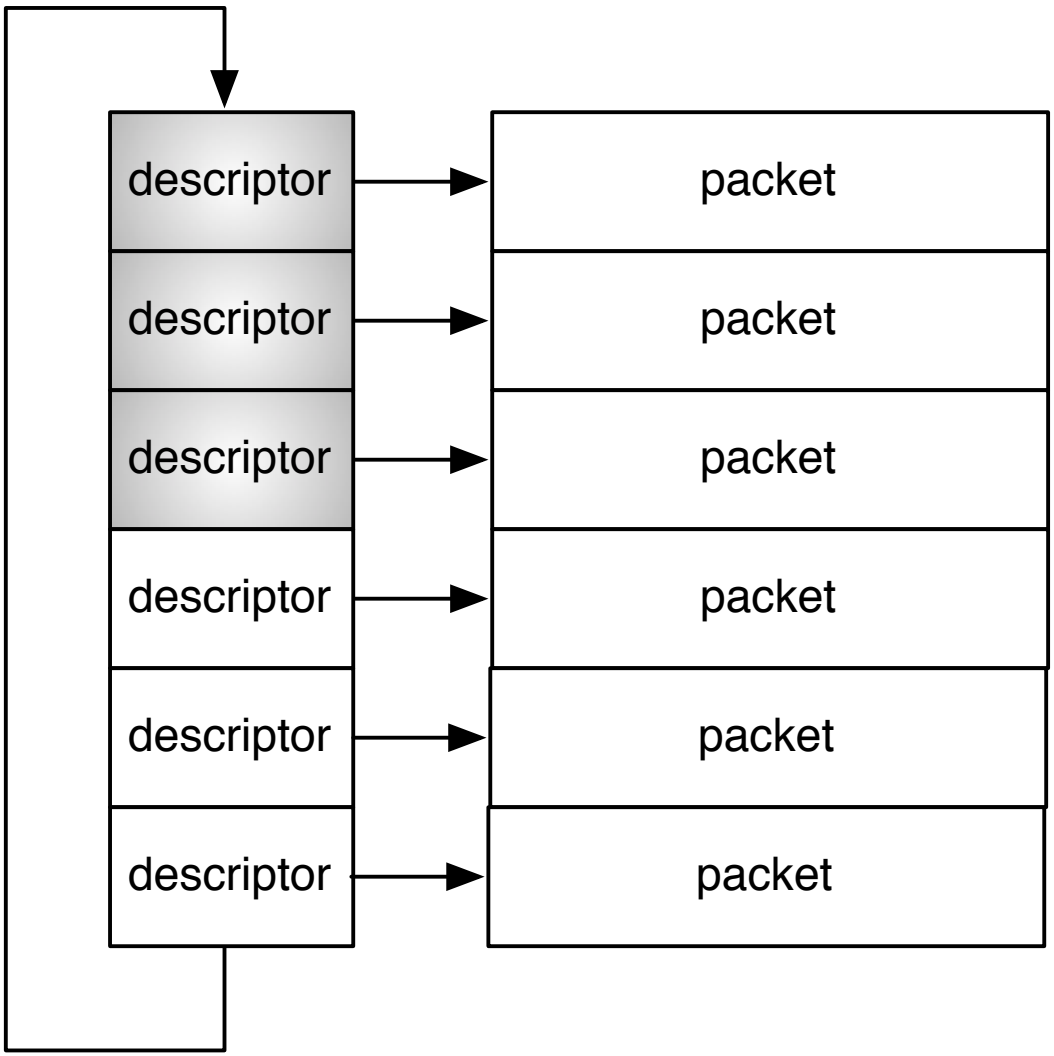
# Receive Ring



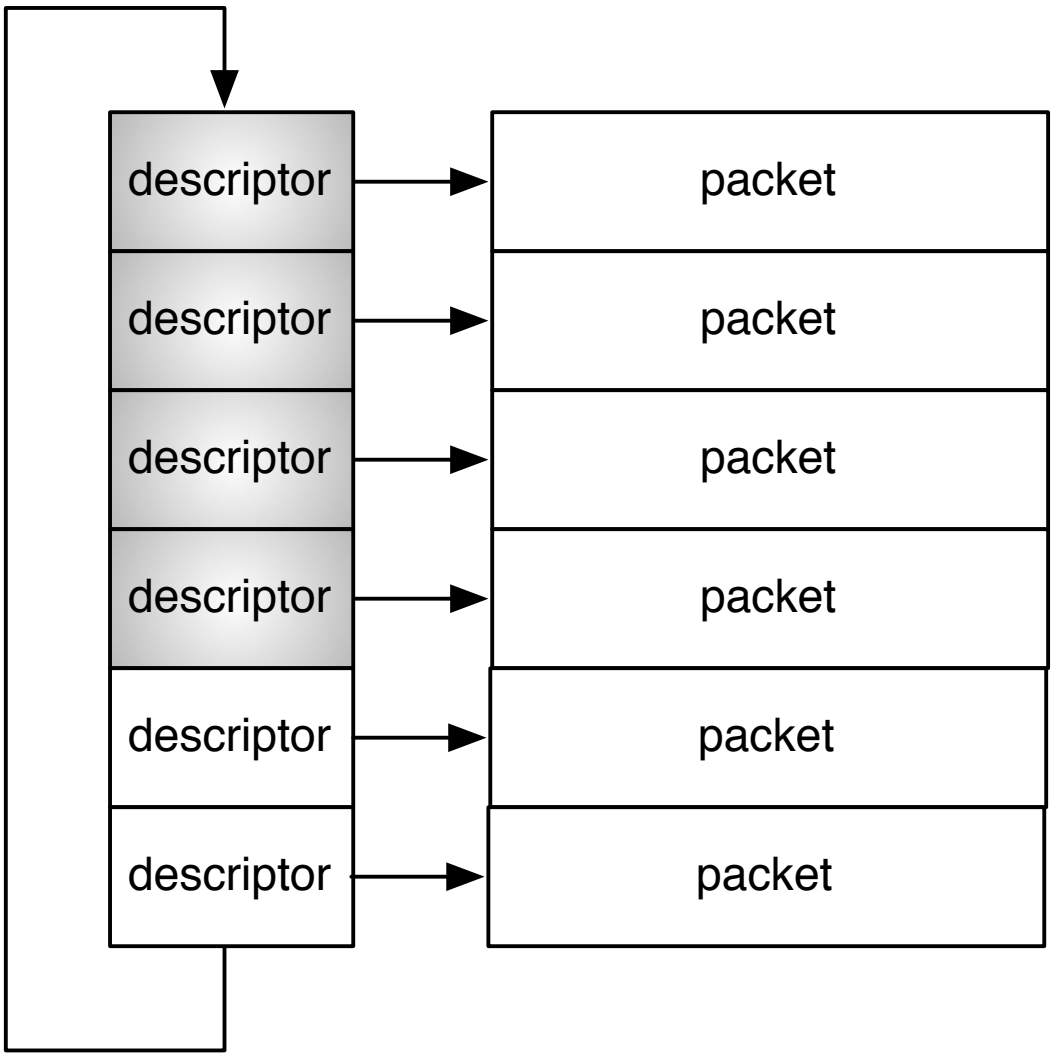
# Receive Ring



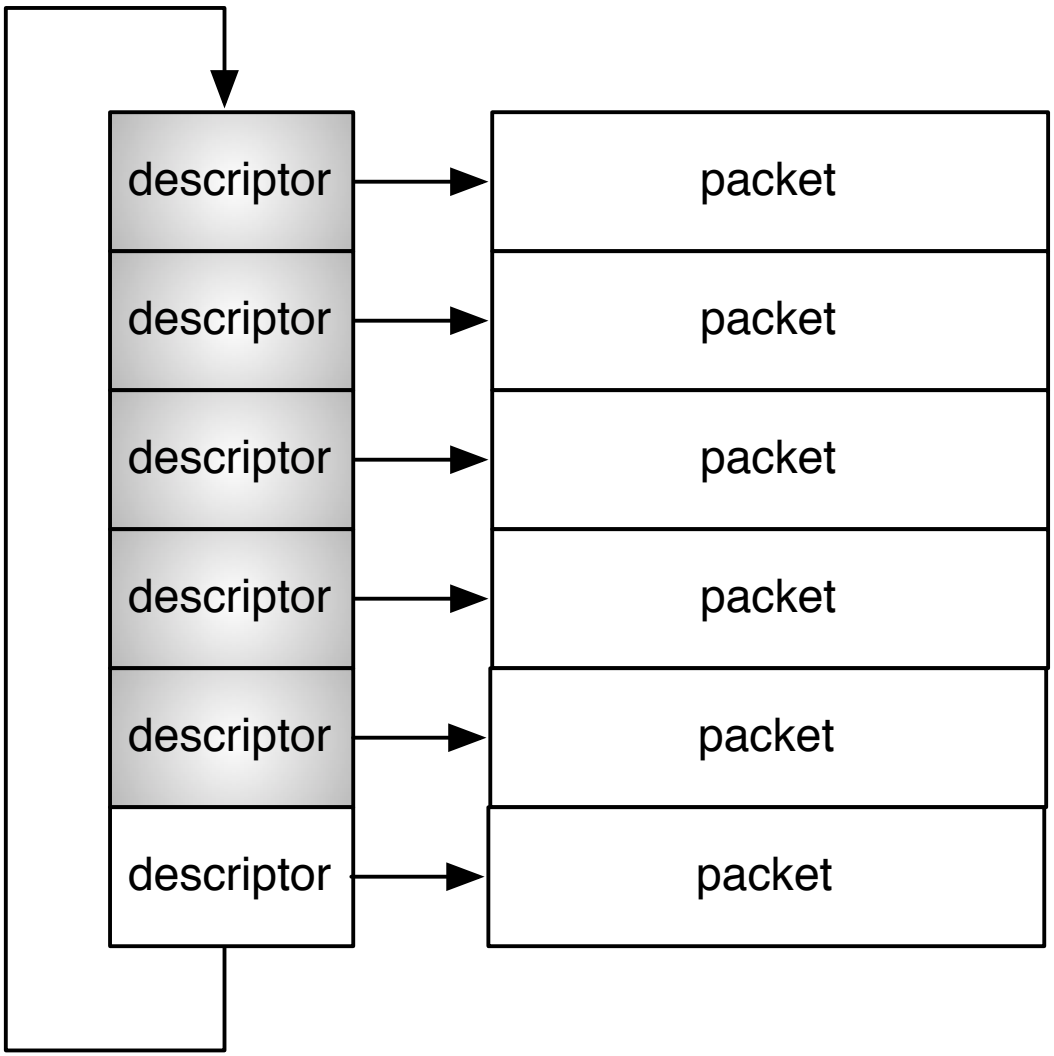
# Receive Ring



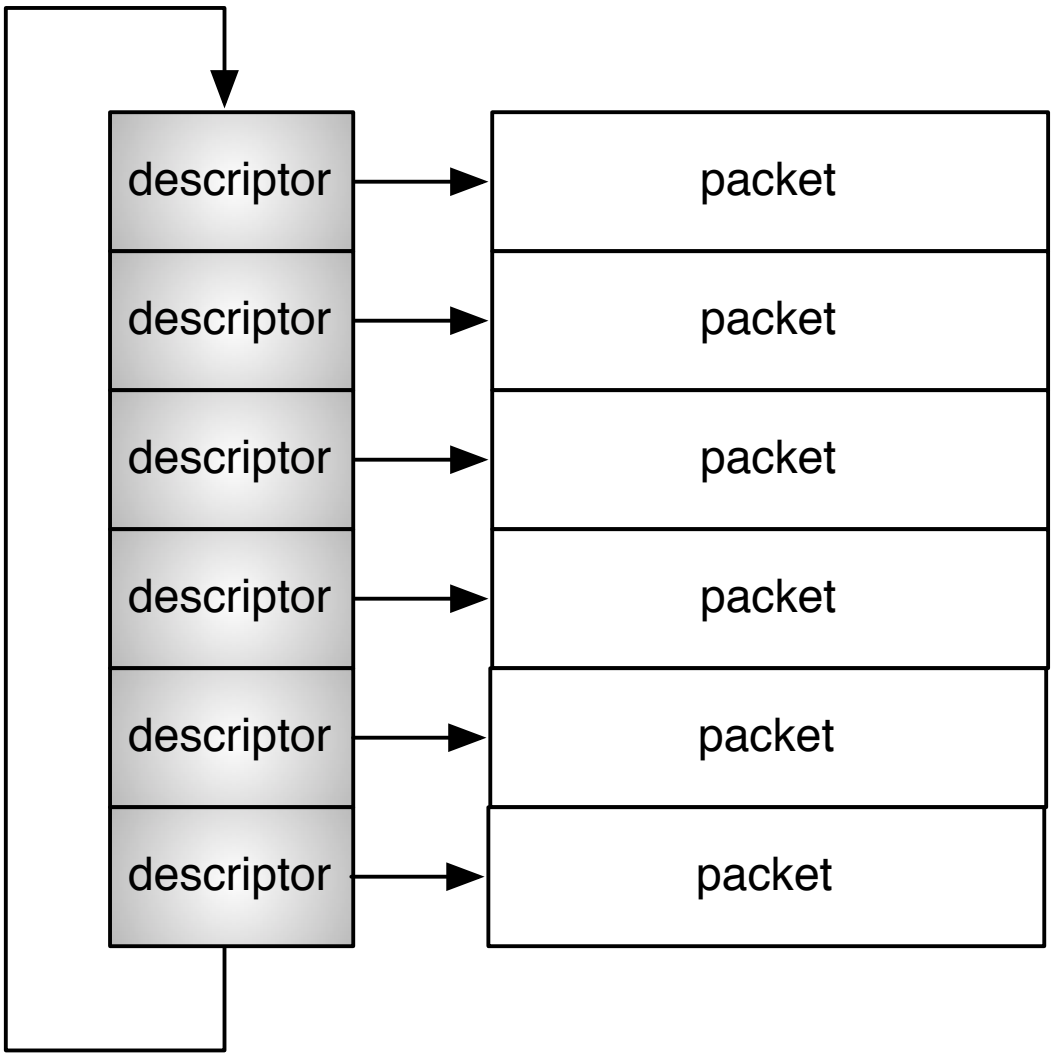
# Receive Ring



# Receive Ring



# Receive Ring



# Transmit Ring

- Looks much like the receive ring
- May have different meta data
- Descriptor still has an ownership bit

# Uses

- Direct access to all packet data
- Speedup for native virtualization (bhyve)
- Single node test systems
- Latency sensitive applications

# VALE Switch

- Virtual Local Ethernet Switch
- Accessed via netmap
- Used to interconnect VMs (qemu etc.)

# Netmap Code Walk Through

# Break

- Please return in 15 minutes

# Netgraph

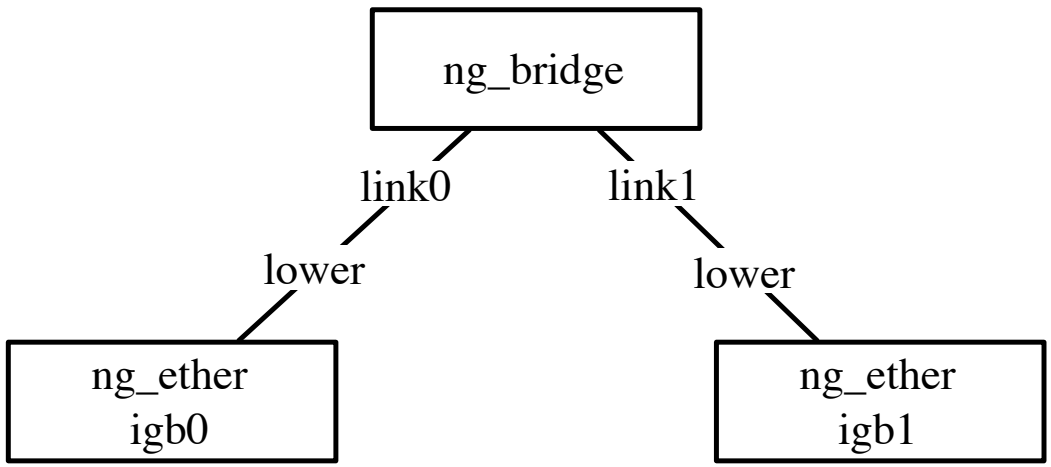
- Directed Graph for Packet Processing
- Used to develop new protocols
- Currently supports
  - PPP
  - ATM
  - Bluetooth

# Object Oriented Packet Processing

Node Class and Object

Hook Method on the Node

# Simple Example: Ethernet Bridge



# Nodes

- Where packet processing is done
- Should only do one thing
  - and do that thing well
- Pass Data
- React to messages

# Some Sample Nodes

| | |
|---|---|
| ng_atm | ATM node |
| ng_bpf | Berkeley Packet Filter |
| ng_bridge | Ethernet bridging |
| ng_device | Device node |
| ng_echo | Echo node |
| ng_etf | Ethertype filtering |
| ng_frame | Frame relay |
| ng_gif | Generic tunnel interface |
| ng_ip | IP input |
| ng_ipfw | Interface between netgraph and IP firewall |
| ng_ksocket | Kernel socket netgraph node type |
| ng_one2many | Multiplexing node |
| ng_split | Separate incoming and outgoing flows |
| ng_tag | Manipulate mbuf tags |
| ng_ubt | Driver for Bluetooth USB devices |
| ng_vlan | IEEE 802.1Q VLAN tagging |

# Node Messages

- Control plane messages
- Configuration
- Setting up connections among nodes

# Node Connections

- One to One
- One to Many
- Many to One

# Hooks

- Form the edges of the graph

# Node Structure

```
struct ng_node {
        char nd_name[NG_NODESIZ]; /* optional globally unique name */
        struct ng_type *nd_type; /* the installed 'type' */
        int nd_flags; /* see below for bit definitions */
        int nd_numhooks; /* number of hooks */
        void    *nd_private; /* node type dependant node ID */
        ng_ID_t nd_ID; /* Unique per node */
        LIST_HEAD(hooks, ng_hook) nd_hooks; /* linked list of node hooks */
        LIST_ENTRY(ng_node)     nd_nodes; /* name hash collision list */
        LIST_ENTRY(ng_node)     nd_idnodes; /* ID hash collision list */
        struct ng_queue    nd_input_queue; /* input queue for locking */
        int nd_refs; /* # of references to this node */
        struct vnet  *nd_vnet; /* network stack instance */
};
```

# Hooks

```
struct ng_hook {
        char hk_name[NG_HOOKSIZ]; /* what this node knows this link as */
        void    *hk_private; /* node dependant ID for this hook */
        int hk_flags; /* info about this hook/link */
        int hk_type; /* tbd: hook data link type */
        struct ng_hook *hk_peer; /* the other end of this link */
        struct ng_node *hk_node; /* The node this hook is attached to */
        LIST_ENTRY(ng_hook) hk_hooks; /* linked list of all hooks on node */
        ng_rcvmsg_t *hk_rcvmsg; /* control messages come here */
        ng_rcvdata_t *hk_rcvdata; /* data comes here */
        int hk_refs; /* dont actually free this till 0 */
};
```

# Netgraph Code Walk Through