Networking

Jiří Benc, Red Hat

KDLP, FI MU

Introduction

- complex topic
- focus on performance
- traditional vs. cloud networking
- BPF (eBPF)

BPF

- bytecode loaded into the kernel
- reaction to events
- not just networking
- innovations × hard to write
- not covered in this lecture

User Point of View

- network interfaces
  - usually having name and numeric ID
  - can be assigned IP addresses
  - can be administratively enabled/disabled, configured, …
- apps operate with IP addresses
  - but can specify an interface
- system tables
  - routing tables
  - neighbor tables
  - …

Basic Packet Processing

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2 → L3 → L4 →
socket lookup → socket queue → app wakeup →
app read → data copy → buffer release

Basic Packet Processing

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2 → L3 → L4 →
socket lookup → socket queue → app wakeup →
app read → data copy → buffer release

app write → data copy → packet descriptor →
L4 → L3 → L2 →
enqueue → dequeue → DMA descriptor → DMA →
tx trigger → NIC tx →
tx IRQ → IRQ handler → memory release

Driver Processing (rx)

NIC rx → DMA

DMA Ring Buffers

- separate tx and rx buffers
- configured by the driver
- contains data and metadata

Driver Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing

Interrupts

- IRQ handler in the driver
- bottom half scheduled
- packet fetched
- new DMA rx buffer allocated

Driver Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP

eXpress Data Path

- BPF program
- called by the driver
- raw packet data
- can drop, redirect or pass on

Driver Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff)

Packet Descriptor

- sk_buff
- allocated by the driver
- packet metadata

Packet Descriptor (sk_buff)

- buffer pointer
- data start
- data length
- header pointers
- incoming/outgoing interface
- L3 protocol
- queue priority
- packet mark
- reference count
- offload fields (vlan tag, hash checksum, ...)

Packet Descriptor (sk_buff)

- buffer pointer
- data start ⬅ allows pop/push
- data length
- header pointers
- incoming/outgoing interface
- L3 protocol
- queue priority
- packet mark
- reference count
- offload fields (vlan tag, hash checksum, ...)

Kernel Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2

Entering the Network Stack

- driver calls helper functions for L2 processing
  - L3 protocol filled in
  - L2 header removed
- handed over to the core kernel

Kernel Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2

Common Handling

- taps on network interface (packet inspection)
- net sched ingress (traffic classification)
  - eBPF hook
- rx hooks (virtual interfaces)
- protocol-independent firewall

Kernel Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2 → L3 → L4

Protocol Layers

- L2 independent
- table of L3 handlers → L3 protocol handler
- L3 header processed and removed
- per-L3 table of L4 handlers → L4 protocol handler
- L4 header processed and removed

Kernel Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2 → L3

L3 – IP

- defragmentation
- routing decision
  - forwarding: skip to tx path
  - local delivery: continue up the stack
- IP firewall (various attachment points)

Kernel Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2 → L3 → L4 →
socket lookup → socket queue → app wakeup

L4 – TCP

- TCP state machine
- socket lookup
- socket enqueue (of the sk_buff)
- application woken up

Kernel Processing (rx)

NIC rx → DMA → rx IRQ →
IRQ handler → schedule processing →
XDP → packet descriptor (sk_buff) →
L2 → L3 → L4 →
socket lookup → socket queue → app wakeup →
app read → data copy → buffer release

Application

- read() syscall
- packet copy
- sk_buff freed

Kernel Processing (tx)

app write → data copy → packet descriptor

Application

- write() syscall
- sk_buff allocation (for DMA)
- data copy

Kernel Processing (tx)

app write → data copy → packet descriptor → L4 → L3

Protocol Layers

- TCP header pushed
- IP header pushed
  - IP firewall
  - routing decision
  - fragmentation (MTU, PMTU)

Kernel Processing (tx)

app write → data copy → packet descriptor → L4 → L3 → L2

Protocol Layers

- L2 header pushed
  - neighbor cache, ARP lookup
- may need to wait for neighbor resolution
  - put to a wait list
  - resumed by incoming ARP reply
  - timer assigned for timeout
  - ICMP signalled back on error

Kernel Processing (tx)

app write → data copy → packet descriptor →
L4 → L3 → L2 →
enqueue → dequeue

Tx Queues

- packet classified and enqueued (tc)
  - eBPF hook
- dequeued
  - based on queue discipline
  - sk_buff priority field
- passed to the driver

Driver Processing (tx)

app write → data copy → packet descriptor →
L4 → L3 → L2 →
enqueue → dequeue → DMA descriptor → DMA →
tx trigger → NIC tx

Pushing to the NIC

- added to tx DMA ring buffer
- signalled to the NIC

Driver Processing (tx)

app write → data copy → packet descriptor →
L4 → L3 → L2 →
enqueue → dequeue → DMA descriptor → DMA →
tx trigger → NIC tx →
tx IRQ → IRQ handler → memory release

Freeing Resources

- NIC signals transmit done
- buffer unmapped, sk_buff released
- counters incremented

XDP-only tx Path

- a separate entry point to the driver
- no sk_buff

Performance Matters!

Performance Problems

- need parallel processing
  - ⇒ multiple hardware queues
  - both rx and tx
  - tx queue mapping: tc

Performance Problems

- packet length unknown in advance
  - ⇒ DMA scatter-gather
  - sk_buff: header + fragmented data
    - complicates packet processing
  - header read needs care
  - header pop may require realloc

Performance Problems

- header push requires realloc
    - ⇒ drivers should reserve header space
    - still may get out of space

Bottlenecks

- stack processing is too heavy
  - ⇒ aggregation of packets
  - processing whole flows
- interrupts are slow
  - ⇒ busy polling under load
- reading memory is slow
  - ⇒ checksum offloading

Checksum Offloading

- FCS (Ethernet checksum) is handled by the NIC
- IP header checksum calculation is cheap
- TCP checksum is expensive
- some protocols (SCTP) use CRC instead

Checksum Offloading

Rx

1. NIC verifies the checksum
2. NIC calculates the checksum
    - preferred

Checksum Offloading

Tx

- checksum on copy from user
- NIC calculates the checksum and fills it in
- the driver may fall back to a software helper

Busy Polling under Load

- NAPI
- the idea:
  - on rx, turn off IRQs
  - fetch packets up to a limit
  - repeat until there are no packets left
  - turn on IRQ
- per rx (hardware) queue
- needs driver support

Aggregation

Rx Aggregation (GRO)

- needs multiple rx queues in NIC
  - configurable filters
- on rx, packets for the same flow from a NAPI batch are combined into a super-packet
  - ⇒ GRO depends on NAPI
  - need to dissect the packets
  - passes the stack as a single packet
  - need to be able to reconstruct the original packets
  - split on tx (GSO)

Aggregation

Tx Aggregation (GSO)

- on tx, a packet is split into smaller packets
  - TCP segmentation for TCP super-packets
  - offloaded to NIC (TSO)
  - ⇒ TSO depends on checksum offloading
  - IP fragmentation for datagram protocols
  - fallback to a software helper when needed

Feature Advertisement

- set of flags in the network interface struct
- administrator can switch them off (ethtool)

A Simple Driver

```
struct my_data {
    struct net_device *dev;
    struct work_struct irq_task;
    ...
};
```

A Simple Driver

```
int fake_bus_probed(int probed_irq) {
    struct net_device *dev = alloc_etherdev(
                    sizeof(struct my_data));
    struct my_data *md = netdev_priv(dev);
    dev->netdev_ops = &my_ops;
    /* Currently active device features */
    dev->features = 0;
    /* User-changeable features */
    dev->hw_features = 0;
    dev->irq = probed_irq;
    md->dev = dev;
    INIT_WORK(&md->irq_task, my_irq_task);
    return register_netdev(dev);
}
```

A Simple Driver

```
static const struct net_device_ops my_ops = {
    .ndo_open            = my_open,
    .ndo_stop            = my_close,
    .ndo_start_xmit      = my_start_xmit,
    .ndo_get_stats       = my_get_stats,
    .ndo_tx_timeout      = my_tx_timeout,
    .ndo_set_mac_address = eth_mac_addr,
    .ndo_validate_addr   = eth_validate_addr,
};
```

A Simple Driver

```
static netdev_tx_t
my_start_xmit(struct sk_buff *skb,
              struct net_device *dev) {
    struct my_data *md = netdev_priv(dev);

    if (!my_do_tx(skb)) {
        netif_stop_queue(dev);
        return NETDEV_TX_BUSY;
    }
    return NETDEV_TX_OK;
}
```

A Simple Driver

```
static int my_open(struct net_device *dev) {
    return request_irq(dev->irq,
                       my_interrupt,
                       0,
                       dev->name, dev);
}
```

A Simple Driver

```
static irqreturn_t
my_interrupt(int irq, void *dev_id) {
    struct net_device *dev = dev_id;
    struct my_data *md = netdev_priv(dev);
    schedule_work(&md.irq_task);
    my_ack_irq(md);
}
```

A Simple Driver

```
static void
my_irq_task(struct work_struct *work) {
    struct my_data *md = container_of(work,
                        struct my_data, irq_task);
    int status = my_get_status(md);
    if (status & MY_TX_COMPLETE) {
        struct sk_buff *skb = my_get_done(md);
        md->dev->stats.tx_bytes += skb->len;
        dev_consume_skb_irq(skb);
        netif_wake_queue(md->dev);
    }
    if (status & MY_RX_COMPLETE)
        my_rx(md->dev);
}
```

A Simple Driver

```
static void my_rx(struct net_device *dev) {
    struct my_buffer *buf = my_rx_buffer(dev);
    struct sk_buff *skb =
        netdev_alloc_skb_ip_align(dev,
                                  buf->len);
    my_fetch_buffer(buf,
                    skb_put(skb, buf->len));
    skb->protocol = eth_type_trans(skb, dev);
    dev->stats.rx_bytes += pkt_len;
    dev->stats.rx_packets++;
    my_free_buffer(buf);
    netif_rx(skb);
}
```