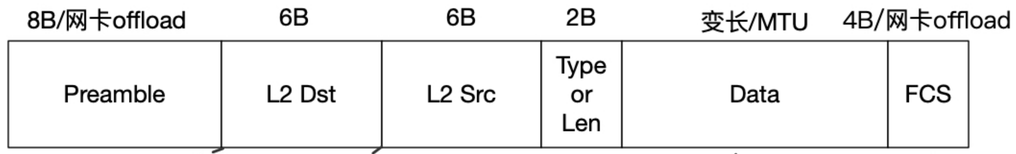
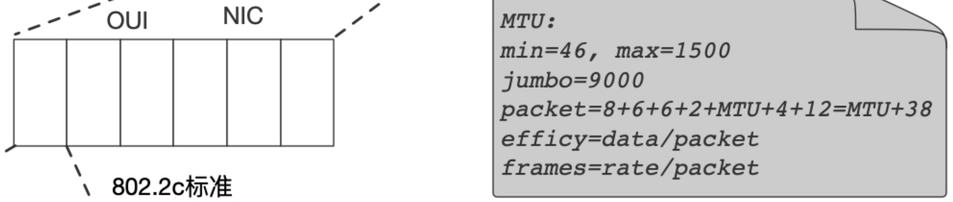
浅谈经典网络

翟增辉 2020.11.18

1. L2 - IEEE802.3 - Frame

802.3 Ethernet II:





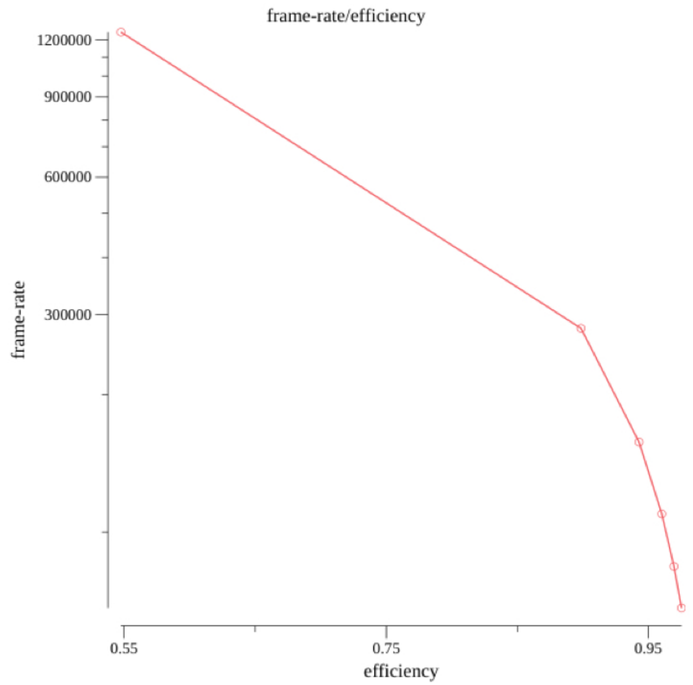
802.2c标准


```
MTU:
min=46, max=1500
jumbo=9000
packet=8+6+6+2+MTU+4+12=MTU+38
effiсy=data/packet
frames=rate/packet
```

```
> ethtool -k rx-fcs on
> ethtool -k rx-all on
```

include/linux/etherdevice.h

```c
/**
 * eth_random_addr - Generate software assigned random Ethernet address
 * @addr: Pointer to a six-byte array containing the Ethernet address
 *
 * Generate a random Ethernet address (MAC) that is not multicast
 * and has the local assigned bit set.
 */
static inline void eth_random_addr(u8 *addr)
{
    get_random_bytes(addr, ETH_ALEN);
    addr[0] &= 0xfe;    /* clear multicast bit */
    addr[0] |= 0x02;    /* set local assignment bit (IEEE802) */
}
```



1. L2 - IEEE802.3 - Jumbo frame

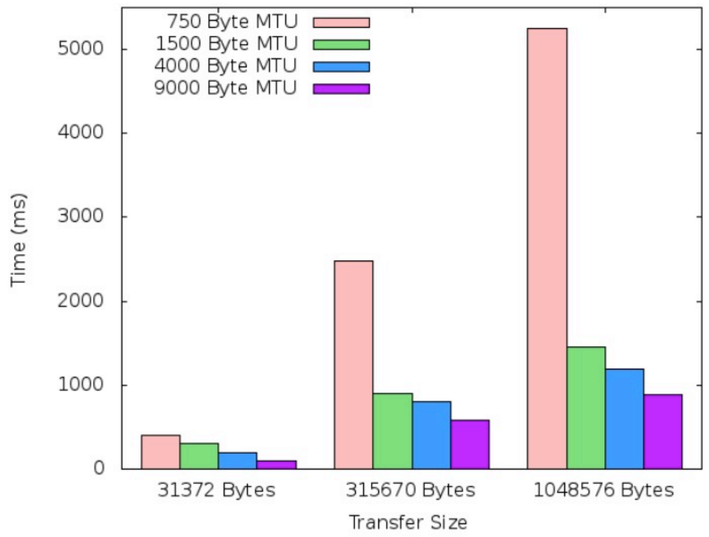


Fig. 6. Completion time of 31372, 315670 and 1048576 byte transfers with a 100ms RTT

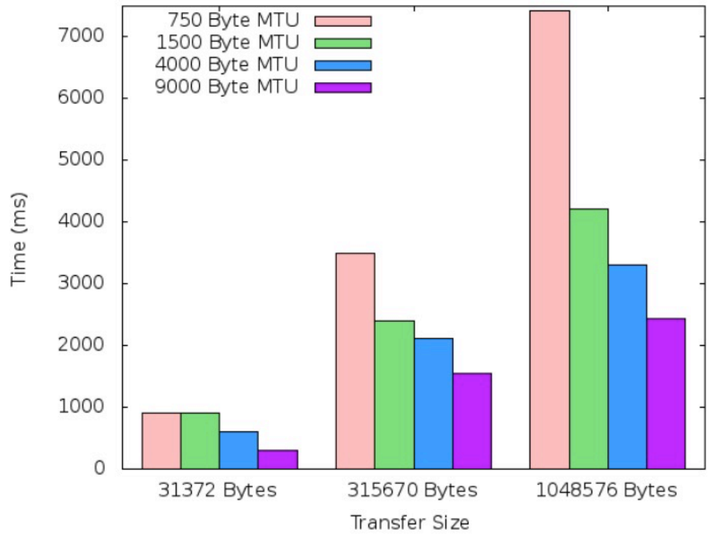
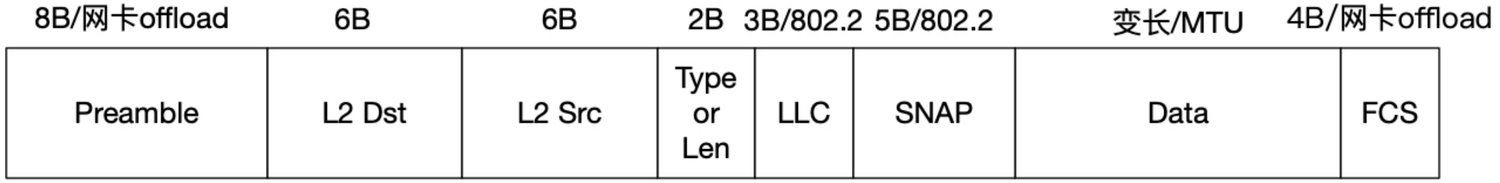


Fig. 7. Completion time of 31372, 315670 and 1048576 byte transfers with a 300ms RTT

- Latency
- Loss
- Throughput
- Frames
- Efficiency
- Cpu loads

1. L2 - IEEE802.2

802.3 Ethernet with SNAP:

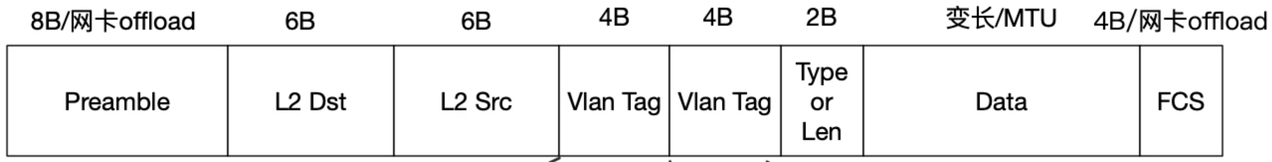


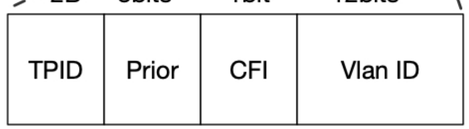
cpf_dpdk/src/cpf_rx.h

```
static int dpdk_parse_pkt(struct rte_mbuf* m)
{
    unsigned short len   = rte_pktmbuf_data_len(m);
    unsigned char* pkt   = rte_pktmbuf_mtod(m, unsigned char*);

    unsigned char flags, proto, ip_type, ip_len, *l3_hdr = pkt + 14;
    unsigned short type = TYPE_eth_unknow, eth_len;
    //IEEE 802.3, DIX Ethernet 2
    unsigned short l3_type = ntohs(*(short *)(void *)(pkt + 12));
    struct pkt_ctx* ctx = (struct pkt_ctx*)m->buf_addr;

    unsigned short frame_len = 14;
    if(unlikely(l3_type < 0x05dc)) {
        // IEEE 802.2, with llc and snap
        l3_type = ntohs(*(short *)(void *)(pkt + 20));
        l3_hdr = pkt + 22;
        frame_len = 22;
    }

    if(l3_type != 0x800) {
        return -1;
    }
```

1. L2 - IEEE802.1q

802.3 Ethernet with SNAP:





设置vlan:

```
> ip link add link eth0 type vlan id 8
> ip link add link eth0 type macvlan mode bridge
```

网卡offload:

```
> ethtool -K rx-vlan on
> ethtool -K tx-vlan on
```

应用程序offload:

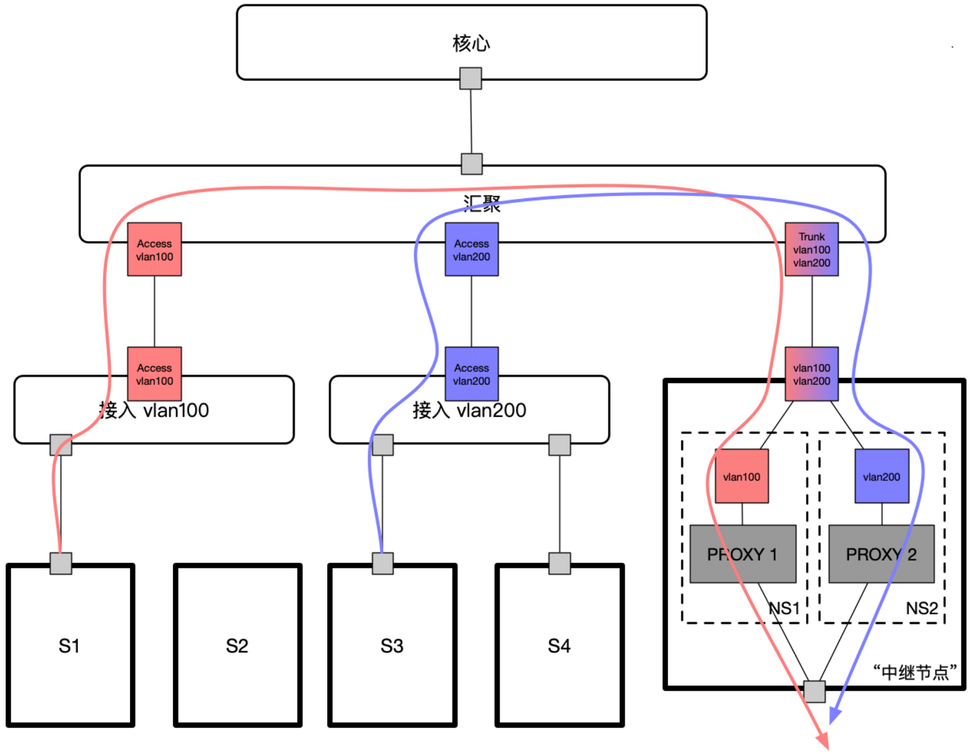
```
type Decoder struct {
    decoders          map[gopacket.LayerType]gopacket.DecodingLayer
    linkLayerDecoder gopacket.DecodingLayer
    linkLayerType     gopacket.LayerType

    sll       layers.LinuxSLL
    lo        layers.Loopback
    eth       layers.Ethernet
    d1q       [2]layers.Dot1Q
    ip4       [2]layers.IPv4
    ip6       [2]layers.IPv6
    icmp4     layers.ICMPv4
    icmp6     layers.ICMPv6
    tcp       layers.TCP
    udp       layers.UDP
    truncated bool
```

H3C交换机侧链路类型:

端口类型	对接收报文的处理		对发送报文的处理
	当接收到的报文不带Tag时	当接收到的报文带有Tag时	
Access端口	为报文添加缺省VLAN的Tag	• 当VLAN ID与缺省VLAN ID相同时，接收该报文 • 当VLAN ID与缺省VLAN ID不同时，丢弃该报文	去掉Tag，发送该报文
Trunk端口	• 当缺省VLAN ID在端口允许通过的VLAN ID列表里时，接收该报文，给报文添加缺省VLAN的Tag • 当缺省VLAN ID不在端口允许通过的VLAN ID列表里时，丢弃该报文	• 当VLAN ID在端口允许通过的VLAN ID列表里时，接收该报文 • 当VLAN ID不在端口允许通过的VLAN ID列表里时，丢弃该报文	• 当VLAN ID与缺省VLAN ID相同，且是该端口允许通过的VLAN ID时：去掉Tag，发送该报文 • 当VLAN ID与缺省VLAN ID不同，且是该端口允许通过的VLAN ID时：保持原有Tag，发送该报文
Hybrid端口			当报文中携带的VLAN ID是该端口允许通过的VLAN ID时，发送该报文，并可以通过**port hybrid vlan**命令配置端口在发送该VLAN（包括缺省VLAN）的报文时是否携带Tag

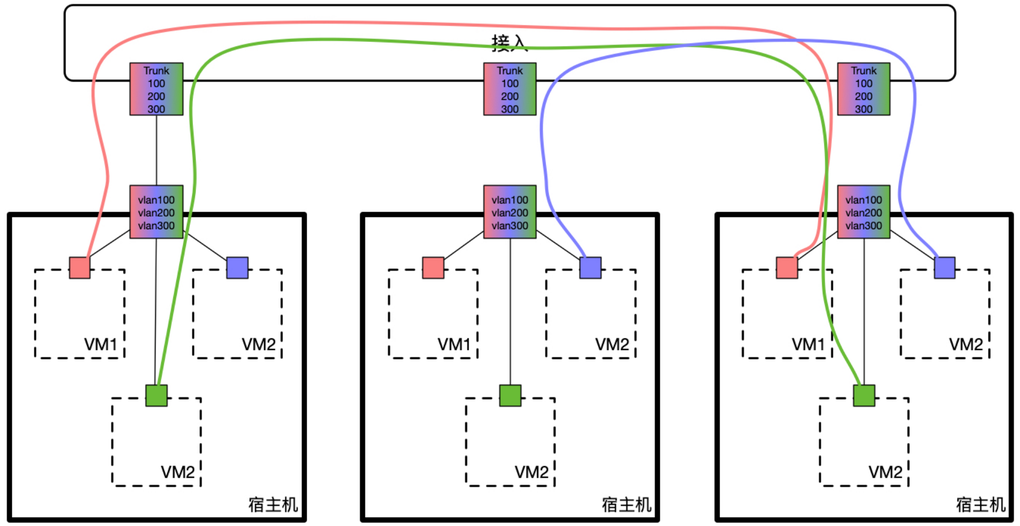
1. L2 - IEEE802.1q - 应用

“中继节点”：



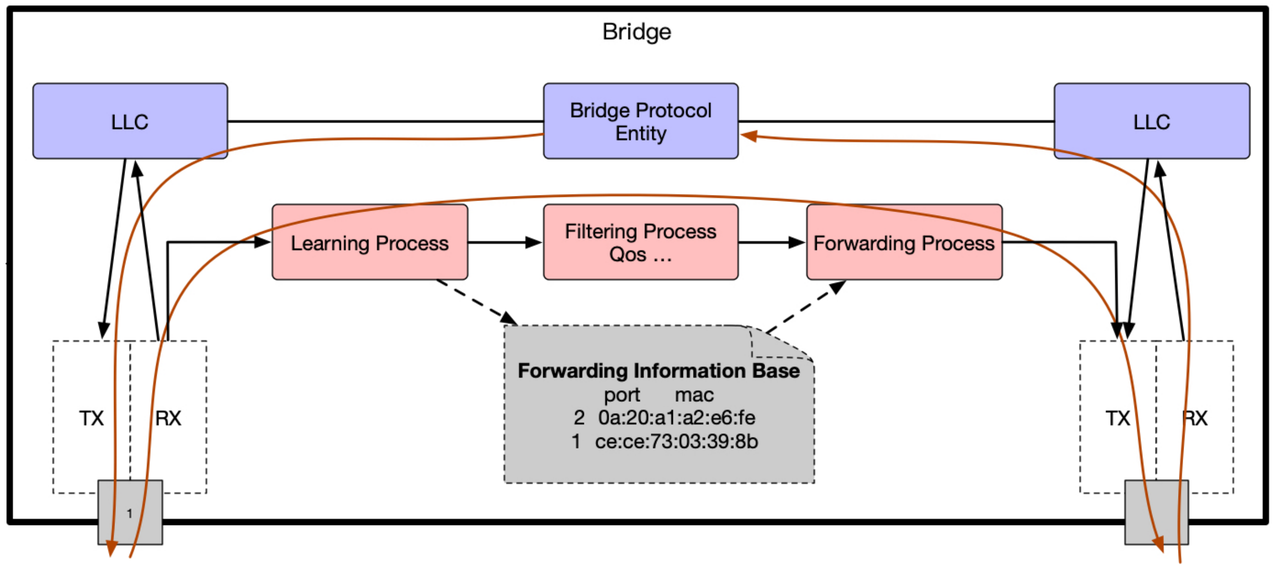
1. 处于所有vlan广播域
2. vlan由linux vlan/macvlan 卸载
3. 使用namespace隔离协议栈

VPC based on vlan：



1. 使用vlan隔离vpc
2. vpc数目受限与vlan个数
3. vm的mac数目受限于交换机缓存

1. L2 - IEEE802.1D Bridging



net/bridge/br_input.c

```c
if (is_broadcast_ether_addr(dest))
    skb2 = skb;
else if (is_multicast_ether_addr(dest)) {
    mdst = br_mdb_get(br, skb, vid);
    if (mdst || BR_INPUT_SKB_CB_MROUTERS_ONLY(skb)) {
        if ((mdst && mdst->mglist) ||
            br_multicast_is_router(br))
            skb2 = skb;
        br_multicast_forward(mdst, skb, skb2);
        skb = NULL;
        if (!skb2)
            goto out;
    } else
        skb2 = skb;

    br->dev->stats.multicast++;
} else if ((dst = __br_fdb_get(br, dest, vid)) &&
        dst->is_local) {
    skb2 = skb;
    /* Do not forward the packet since it's local. */
    skb = NULL;
}

if (skb) {
    if (dst) {
        dst->used = jiffies;
        br_forward(dst->dst, skb, skb2);
    } else
        br_flood_forward(br, skb, skb2);
}
```

查看转发表：

```
H3C:
> display mac-address
```

```
Linux bridge:
> brutal showmacs br0
```

```
OpenVSwitch:
> ovs-appctl fdb/show
```

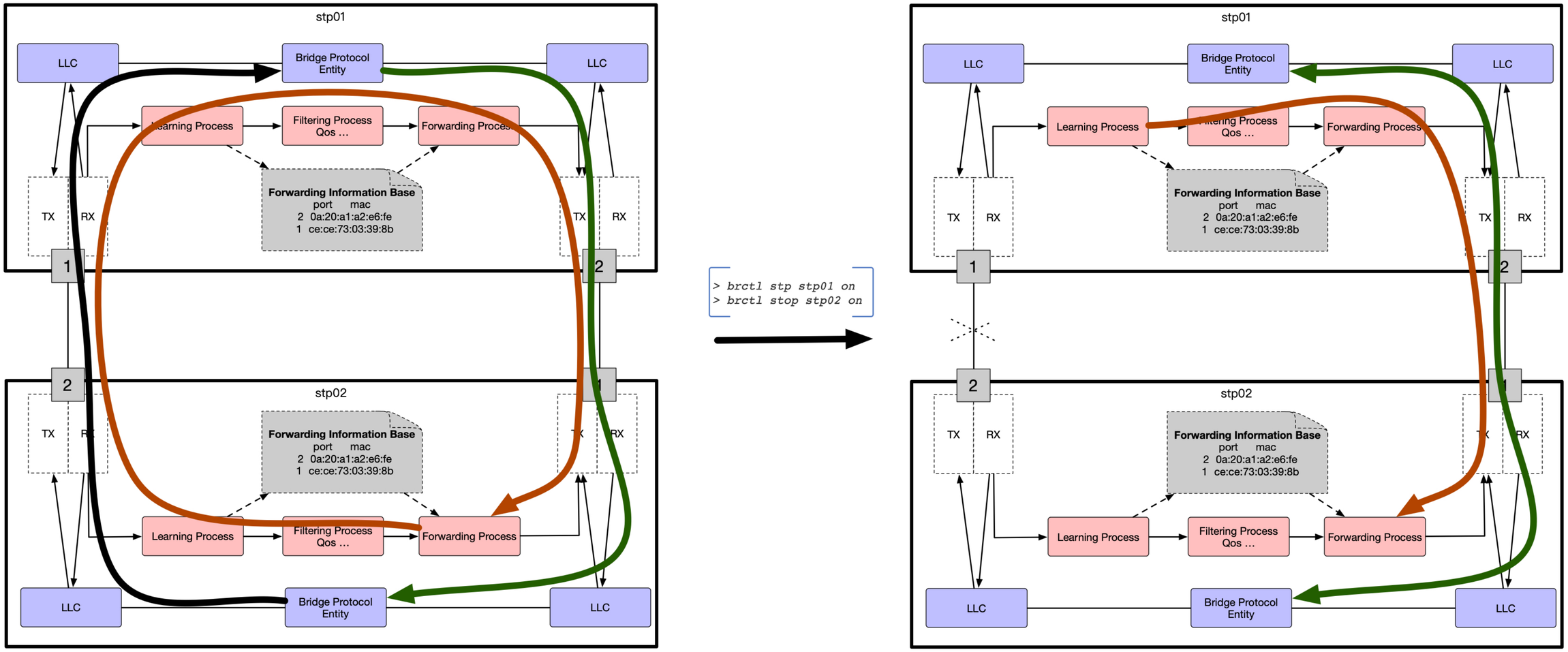
802.1D标准:

K.2 Frame duplication

A unicast frame whose destination address has been learned by the Bridges that can forward it, can only be buffered for transmission on one Port of one Bridge at any one time, and cannot be duplicated.

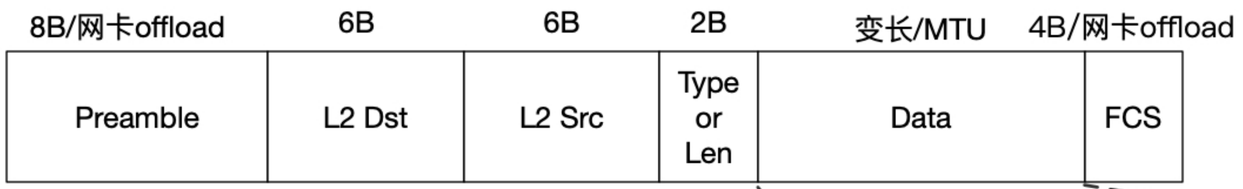
A unicast frame whose destination address has not be learned, can be flooded by some Bridges, and therefore buffered on multiple outbound Ports at the same time, and can be duplicated during network reconfiguration. Figure K-1 provides an example. In this network fragment, the Root Bridge is assumed to be somewhere to the left of Bridge A's Port A1, and the Port Path Costs of the three Bridges result in the active topology shown, with Port B3 Discarding and the remaining Ports all Forwarding.

1. L2 - IEEE802.1D STP



1. L3 - ARP - 标准

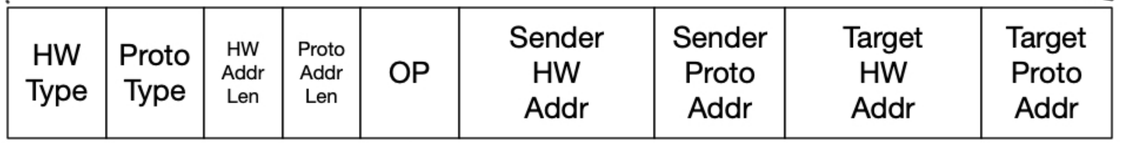
802.3 Ethernet II:



0x806:

```
switch (l3_type) {
case 0x800:
    ctx->src_ip = htonl(*(unsigned int *)(void *)(l3_hdr + 12));
    ctx->dst_ip = ntohl(*(unsigned int *)(void *)(l3_hdr + 16));
    //过滤非资产ip
    if (g_filter_accept_enable)  {
        if (!flume_ip_in_range(ctx->dst_ip, &ip_range_accept) &&
            (!flume_ip_in_range(ctx->src_ip, &ip_range_accept))) {
            continue;
        }
    }

    if (g_filter_drop_enable) {
        if (flume_ip_in_range(ctx->src_ip, &ip_range_drop) || (flume_ip_in_range(ctx->dst_ip, &ip_range_drop))) {
            continue;
        }
    }

case 0x806:

    if (NULL == (pkt = rte_pktmbuf_alloc(pktmbuf_pool[socket_id]))) {
        continue;
    }

    rte_pktmbuf_attach(pkt, pkts[i]);
    pkts_copy[j] = pkt;
    j++;
    break;
```

RFC 826 ARP:



Request: 1
Reply: 2

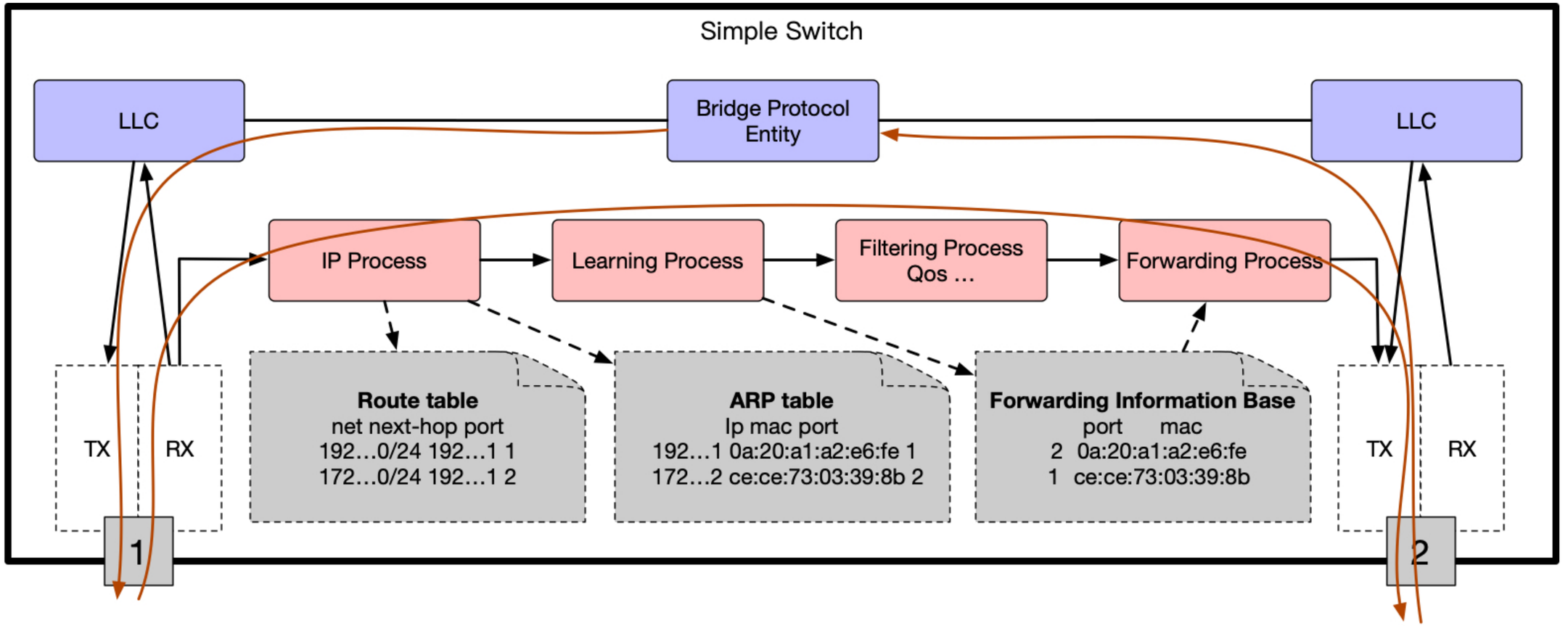
H3C:
> display arp

Linux:
> ip neigh show
> arp -n

发送时:

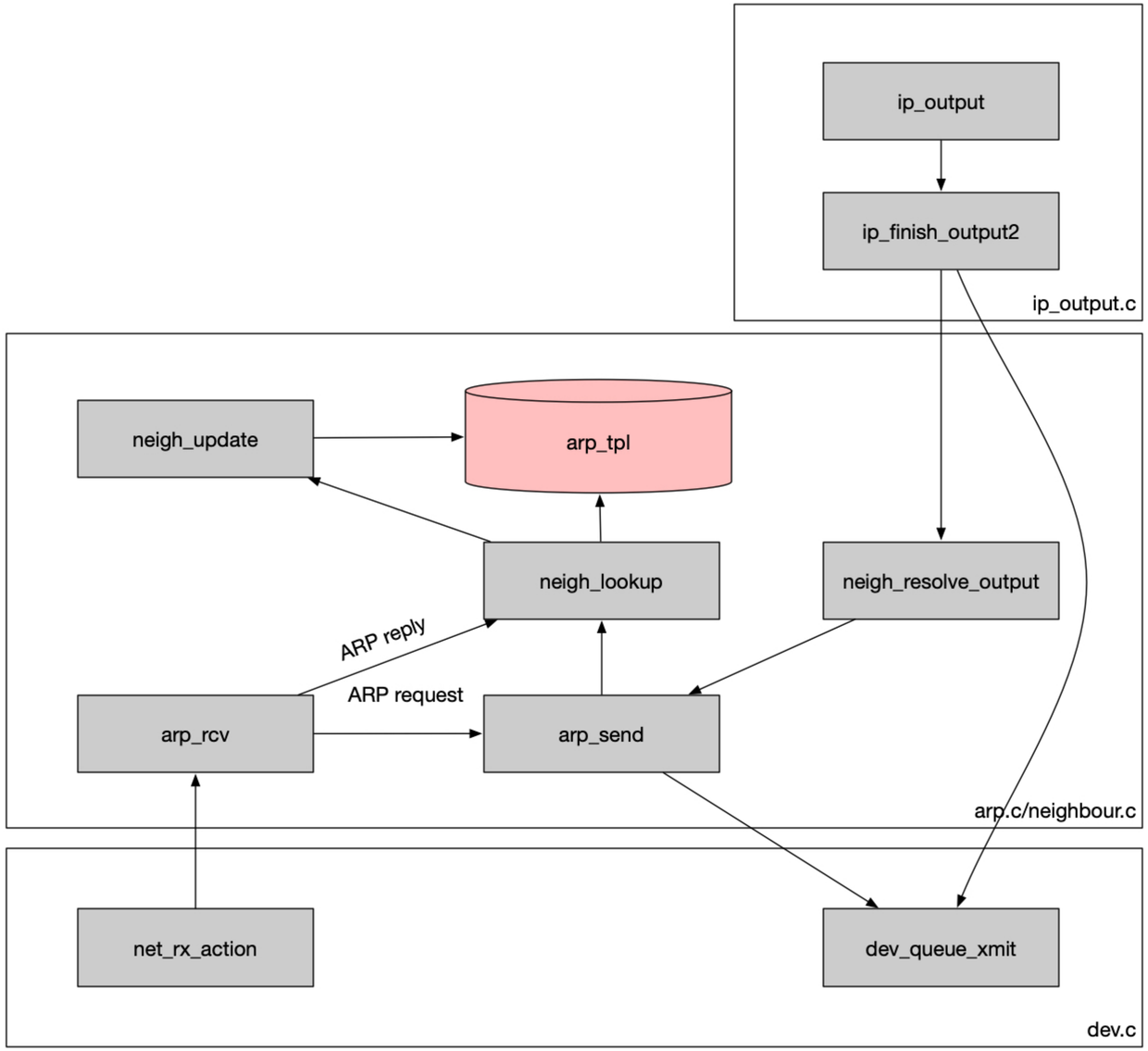
1. 系统维护本地ARP表，存储<mac地址，ip地址>映射关系
2. 发送消息时，先找到下一跳，非本地网络下一跳为网关ip地址，mac地址为网关mac，发送
3. 本地网络下一跳直接为目的ip地址，在ARP表搜索目标ip，找到则发送
4. 找不到则ARP Request该地址，等待回应，收到后更新ARP表，继续第3步发送
5. 收不到ARP Reply则发送失败，丢弃包

1. L3 - ARP - 交换机



1. ARP表，包含<ip, mac>对应关系
2. FIB表，包含<mac, port>对应关系
3. 实现了IEEE802.1d、rfc826、rfc791协议

1. L3 - ARP - Linux实现&特性

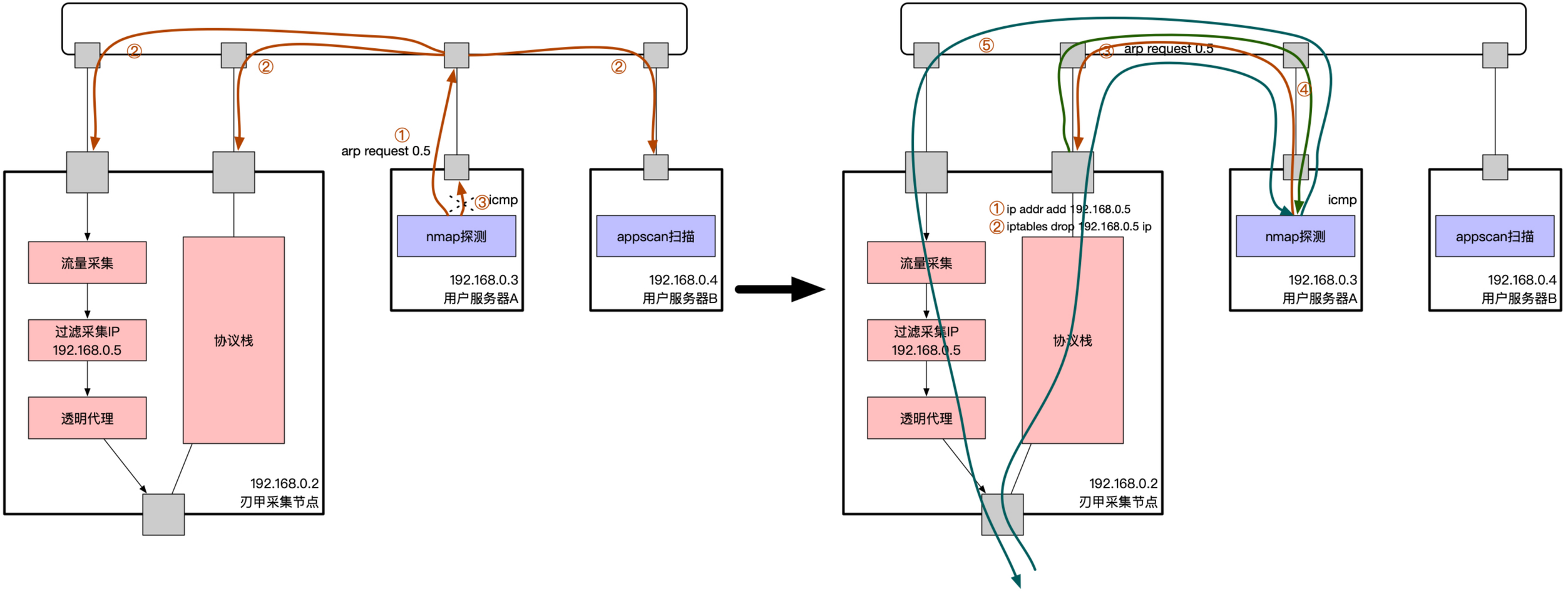


> sysctl -a | grep neigh
anycast_delay=..
proxy_delay=...
proxy_qlen=...

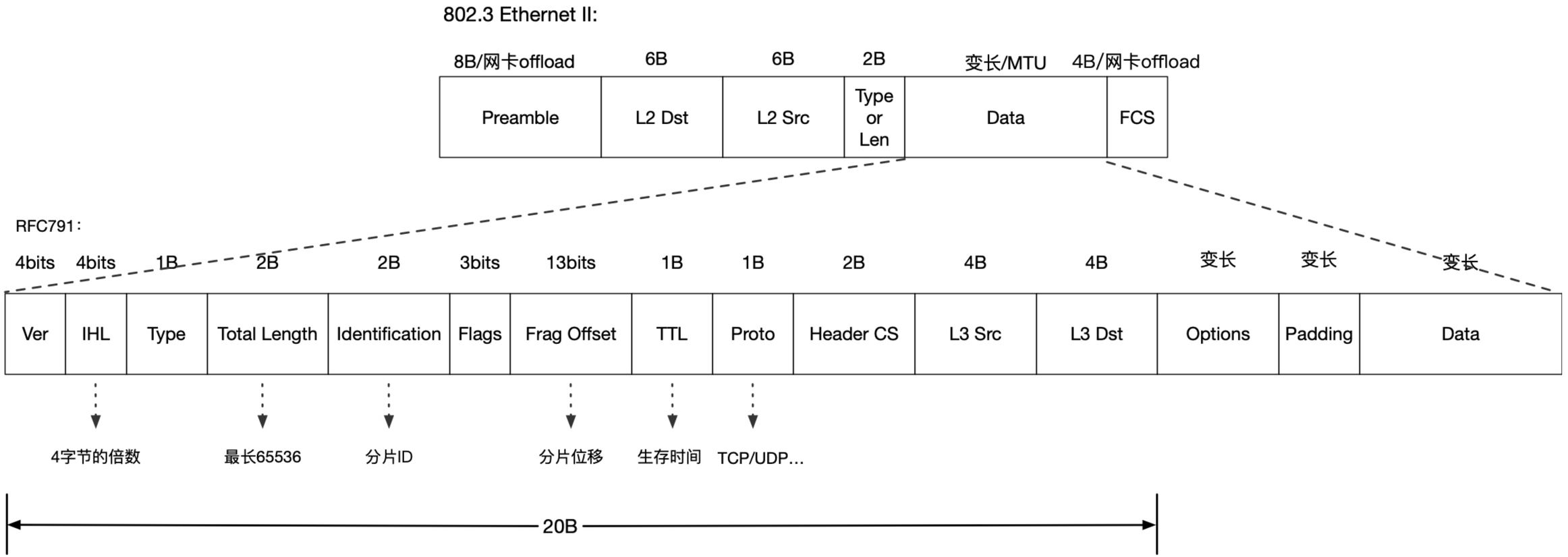
> man 7 arp

> sysctl -a | grep arp
arp_accept=...
arp_announce=...
arp_filter=...
arp_notify=...

1. L3 - ARP - HW场景

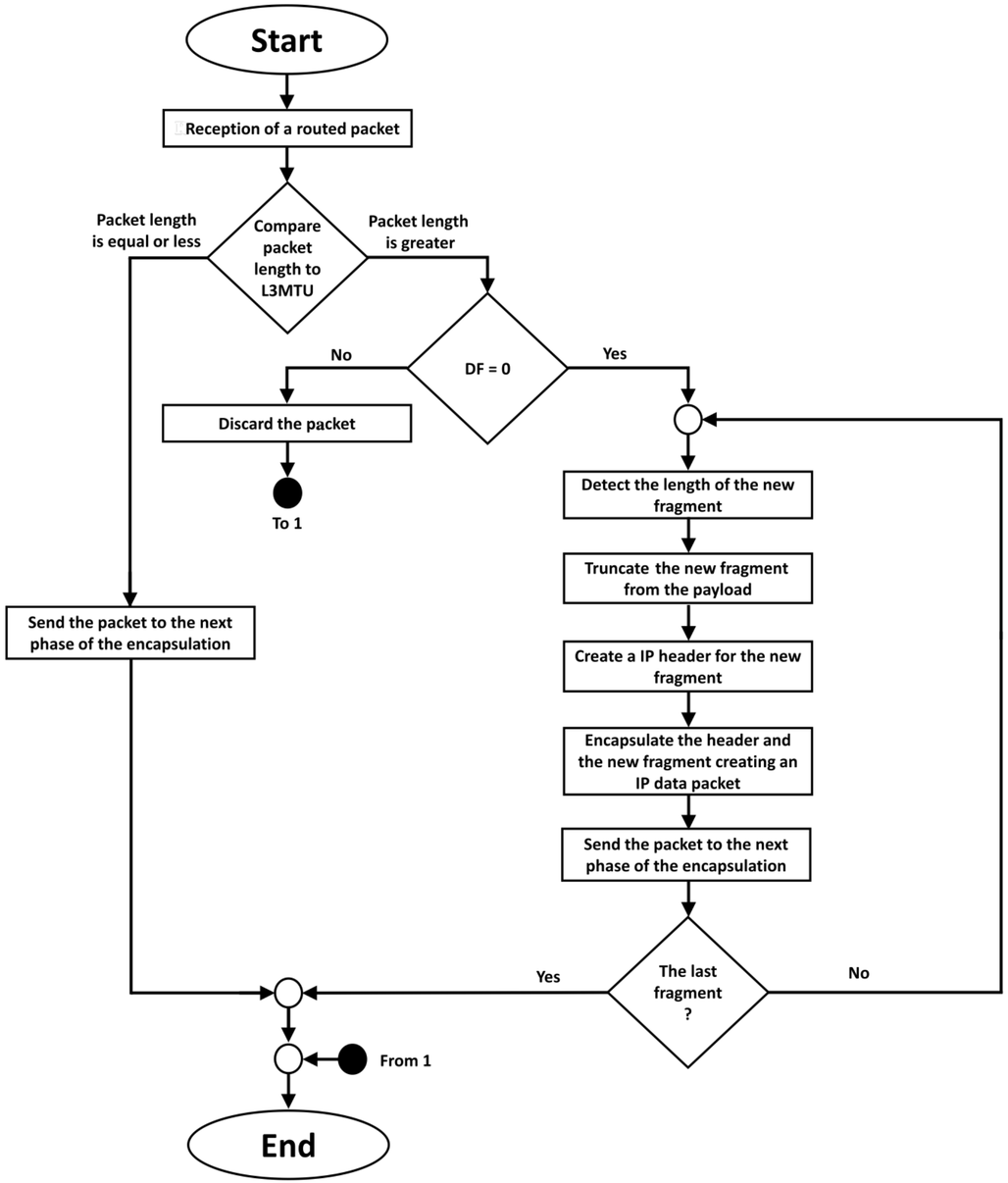

②
②
②
①
arp request 0.5
③icmp
nmap探测
192.168.0.3
用户服务器A
appscan扫描
192.168.0.4
用户服务器B
流量采集
过滤采集IP
192.168.0.5
透明代理
协议栈
192.168.0.2
刃甲采集节点
⑤
③ arp request 0.5
④
① ip addr add 192.168.0.5
② iptables drop 192.168.0.5 ip
icmp
nmap探测
192.168.0.3
用户服务器A
appscan扫描
192.168.0.4
用户服务器B
流量采集
过滤采集IP
192.168.0.5
透明代理
协议栈
192.168.0.2
刃甲采集节点

2. L3 - IP - Format



2. L3 - IP

分片算法：



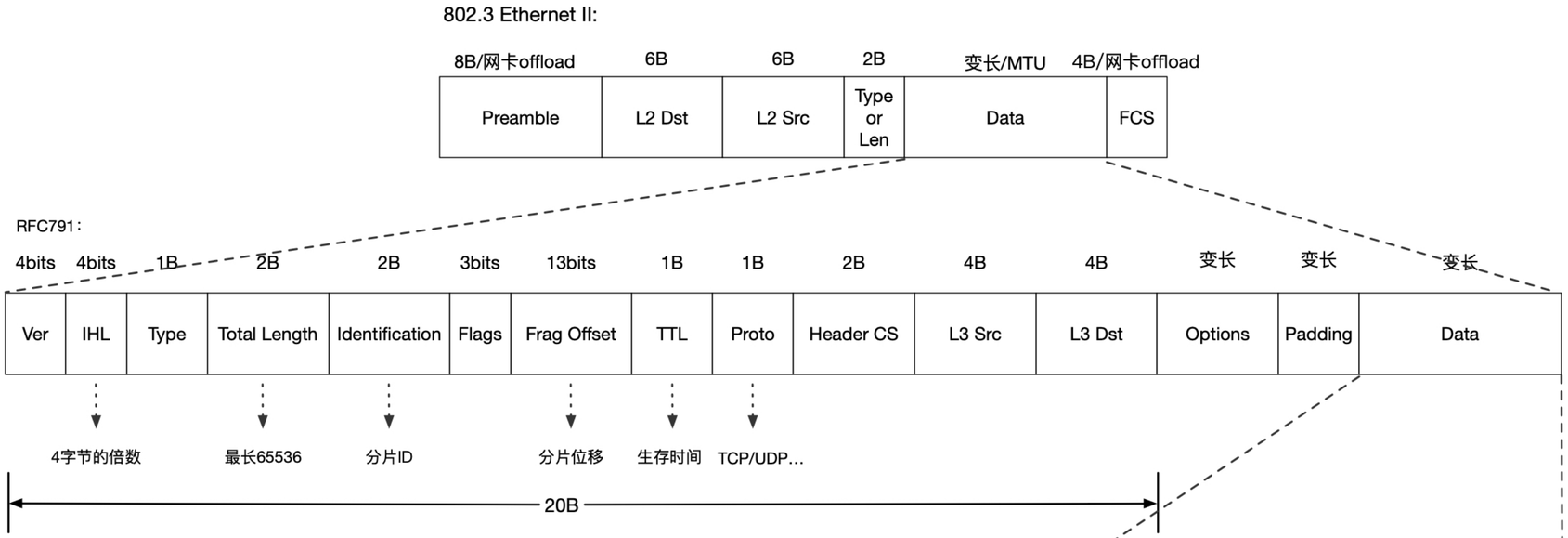
Linux实现相关：

1. net/ipv4/ip_fragment.c
2. ethtool -k eth0 gso on 网卡offload
3. sysctl -a | grep ipfrag调优时间、大小等

其他特性：

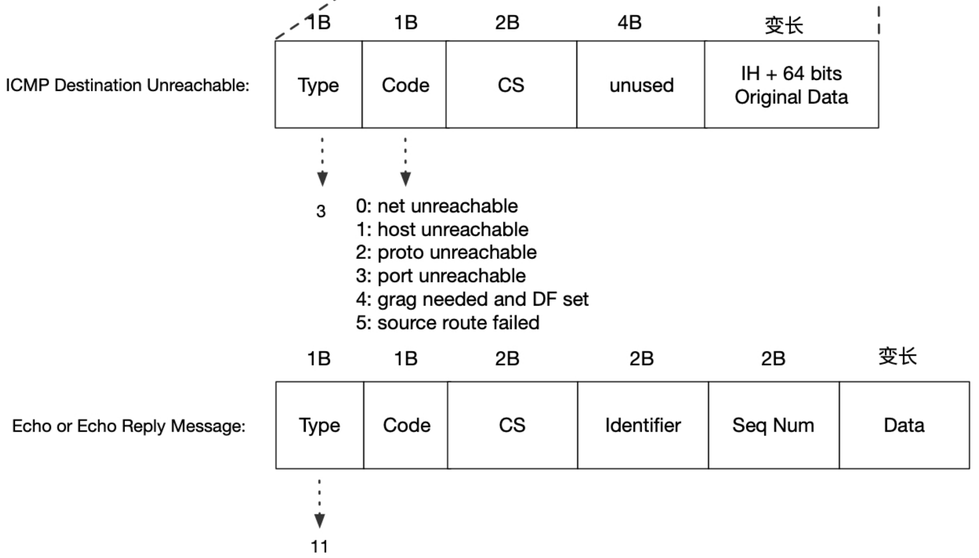
1. ttl，每个接收IP包的模块必须要ttl-1
2. rfc6890定义特殊IP地址

2. L3 - ICMP - Format

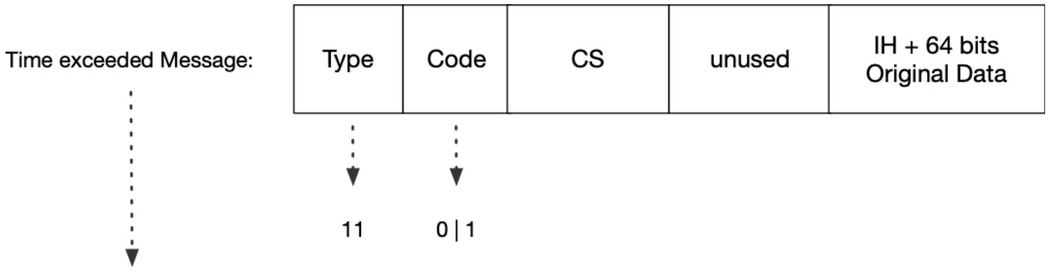


RFC1122:

```
A host SHOULD generate Destination Unreachable messages with
code:

2     (Protocol Unreachable), when the designated transport
      protocol is not supported; or

3     (Port Unreachable), when the designated transport
      protocol (e.g., UDP) is unable to demultiplex the
      datagram but has no protocol mechanism to inform the
      sender.

A Destination Unreachable message that is received MUST be
reported to the transport layer.  The transport layer SHOULD
use the information appropriately; for example, see Sections
4.1.3.3, 4.2.3.9, and 4.2.4 below.  A transport protocol
that has its own mechanism for notifying the sender that a
port is unreachable (e.g., TCP, which sends RST segments)
MUST nevertheless accept an ICMP Port Unreachable for the
same purpose.

A Destination Unreachable message that is received with code
0 (Net), 1 (Host), or 5 (Bad Source Route) may result from a
routing transient and MUST therefore be interpreted as only
a hint, not proof, that the specified destination is
unreachable [IP:11].  For example, it MUST NOT be used as
proof of a dead gateway (see Section 3.3.1).
```



2. L3 - ICMP - Traceroute



Description

If the gateway processing a datagram finds the time to live field

[Page 6]

September 1981
RFC 792

is zero it must discard the datagram. The gateway may also notify the source host via the time exceeded message.

If a host reassembling a fragmented datagram cannot complete the reassembly due to missing fragments within its time limit it discards the datagram, and it may send a time exceeded message.

If fragment zero is not available then no time exceeded need be sent at all.

Code 0 may be received from a gateway. Code 1 may be received from a host.

```
[root@localhost ~]# traceroute -I 39.156.69.79
traceroute to 39.156.69.79 (39.156.69.79), 30 hops max, 60 byte packets
 1  * * *
 2  115.236.5.57 (115.236.5.57)  3.971 ms  4.070 ms  4.155 ms
 3  61.164.4.102 (61.164.4.102)  2.727 ms  2.716 ms  2.813 ms
 4  * * 61.164.8.122 (61.164.8.122)  6.566 ms
 5  * * *
 6  202.97.17.90 (202.97.17.90)  35.669 ms  33.100 ms  33.282 ms
 7  * * *
 8  * * *
 9  111.24.2.245 (111.24.2.245)  35.558 ms *  36.325 ms
10  * 111.24.14.6 (111.24.14.6)  33.578 ms  33.561 ms
11  111.13.188.38 (111.13.188.38)  33.649 ms  33.654 ms  34.998 ms
12  39.156.27.1 (39.156.27.1)  33.559 ms  33.599 ms  33.650 ms
13  * * *
14  * * *
15  * * *
16  * * *
17  39.156.69.79 (39.156.69.79)  34.532 ms  34.576 ms  34.526 ms
```

2. L3 - ICMP - Unreachable

net/ipv4/icmp.c

```
    /*
     *  Throw it at our lower layers
     *
     *  RFC 1122: 3.2.2 MUST extract the protocol ID from the passed
     *       header.
     *  RFC 1122: 3.2.2.1 MUST pass ICMP unreach messages to the
     *       transport layer.
     *  RFC 1122: 3.2.2.2 MUST pass ICMP time expired messages to
     *       transport layer.
     */

    /*
     *  Check the other end isn't violating RFC 1122. Some routers send
     *  bogus responses to broadcast frames. If you see this message
     *  first check your netmask matches at both ends, if it does then
     *  get the other vendor to fix their kit.
     */

    if (!net->ipv4.sysctl_icmp_ignore_bogus_error_responses &&
        inet_addr_type(net, iph->daddr) == RTN_BROADCAST) {
        net_warn_ratelimited("%pI4 sent an invalid ICMP type %u, code %u
                     &ip_hdr(skb)->saddr,
                     icmph->type, icmph->code,
                     &iph->daddr, skb->dev->name);
        goto out;
    }

    icmp_socket_deliver(skb, info);

out:
    return;
```

ping_err.stp

```
global count=0

probe kernel.function("ping_err") {
    count++
        if (count % 5 == 0)
            printf( "sys_sync called %d times\n", count);
}

probe timer.ms(10000){
    printf(" %d times\n\n",count);
}
```

> netstat -s | grep “Icmp:” -A 5

```
[root@localhost ~]# netstat -s | grep "Icmp:" -A 5
Icmp:
    692039 ICMP messages received
    34932 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 69177
        timeout in transit: 180
```

2. L3 - Route

直连路由：

1. 配置ip/cidr的网卡直接连接在设备上出现的路由
2. display ip routing-table protocol direct
3. ip route show scope link

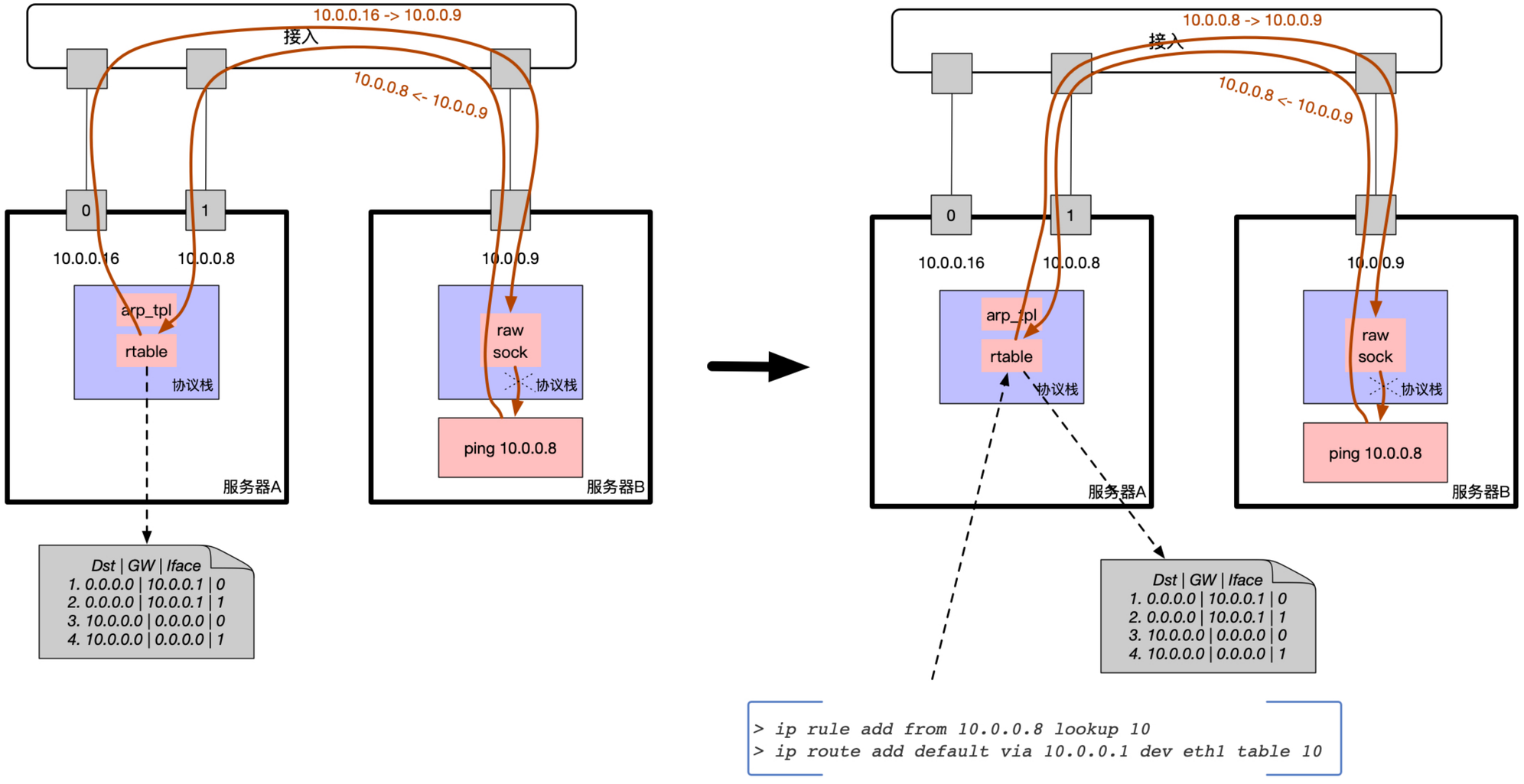
静态路由：

1. 手动配置的路由，依赖网络管理员
2. 策略路由、根据策略挑选的路由

动态路由：

1. 动态路由协议生成的路由
2. 链路状态协议：ospf（Dijkstra算法）
3. 距离矢量协议：rip等
4. 路径矢量协议：bgp等

2. L3 - Route - 策略路由场景



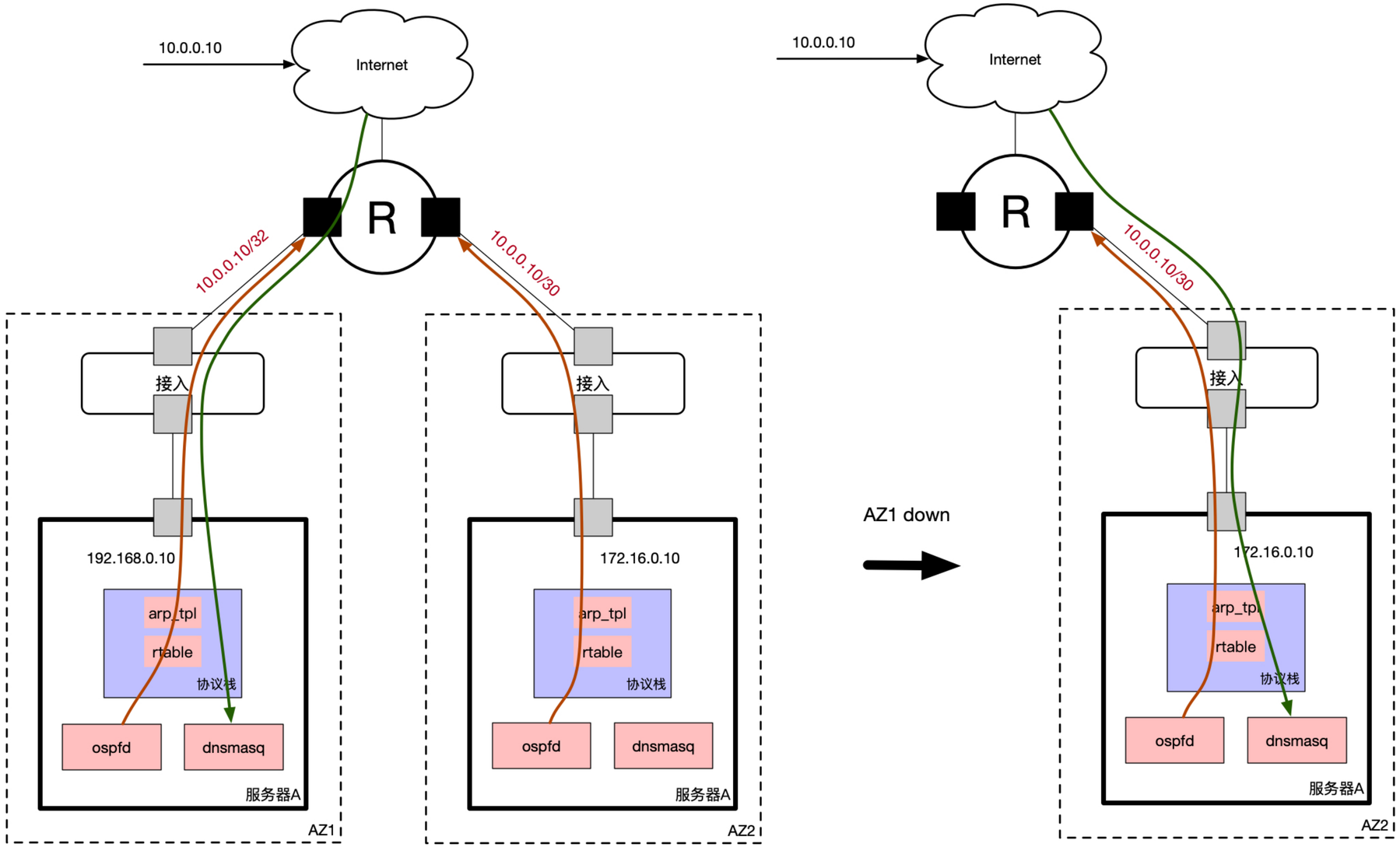

```
> ip rule add from 10.0.0.8 lookup 10
> ip route add default via 10.0.0.1 dev eth1 table 10
```

2. L3 - Route - 选择策略

Example 4.4. Routing Selection Algorithm in Pseudo-code

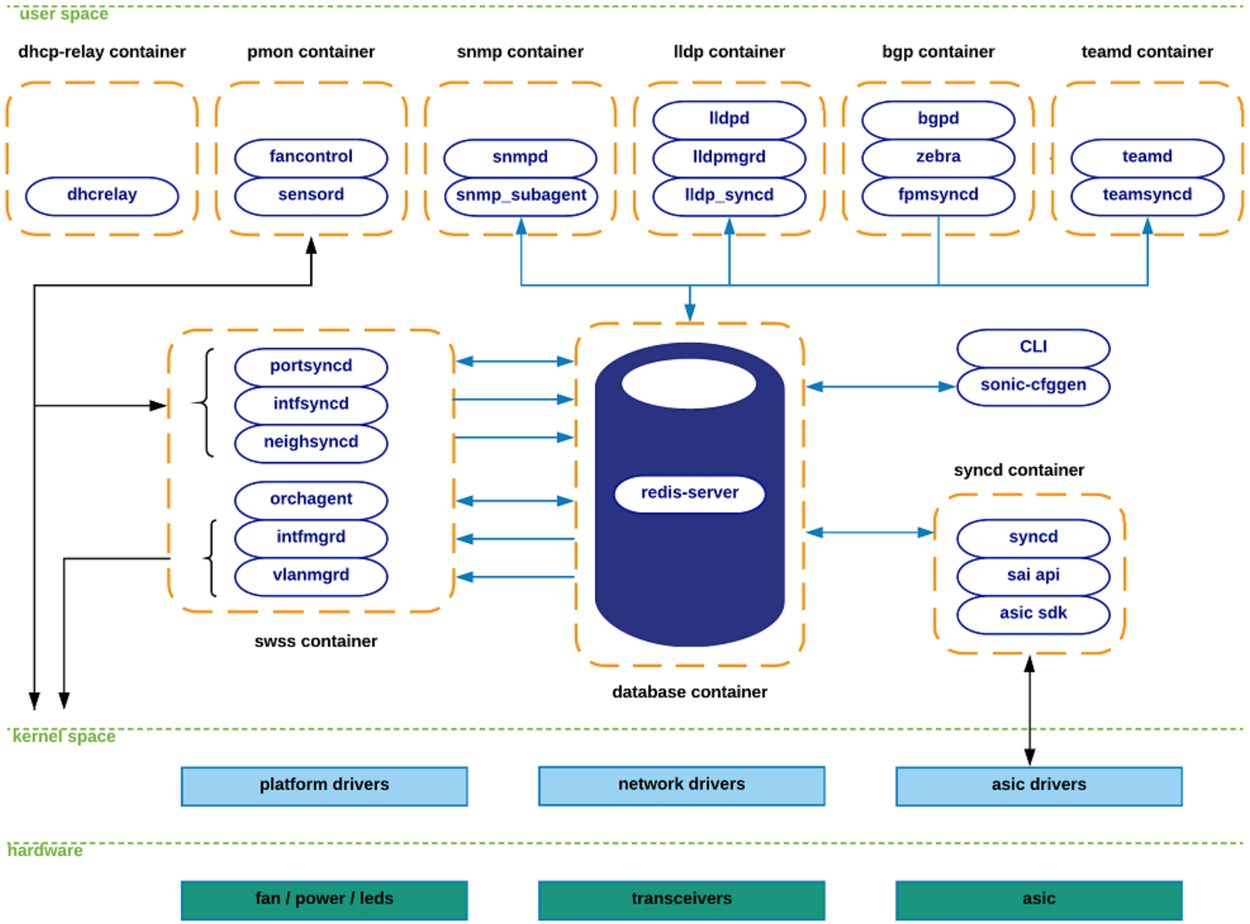
```
if packet.routeCacheLookupKey in routeCache :
    route = routeCache[ packet.routeCacheLookupKey ]
else
    for rule in rpdb :
        if packet.rpdbLookupKey in rule :
            routeTable = rule[ lookupTable ]
            if packet.routeLookupKey in routeTable :
                route = route_table[ packet.routeLookup_key ]
```

2. L3 - Route - ospf



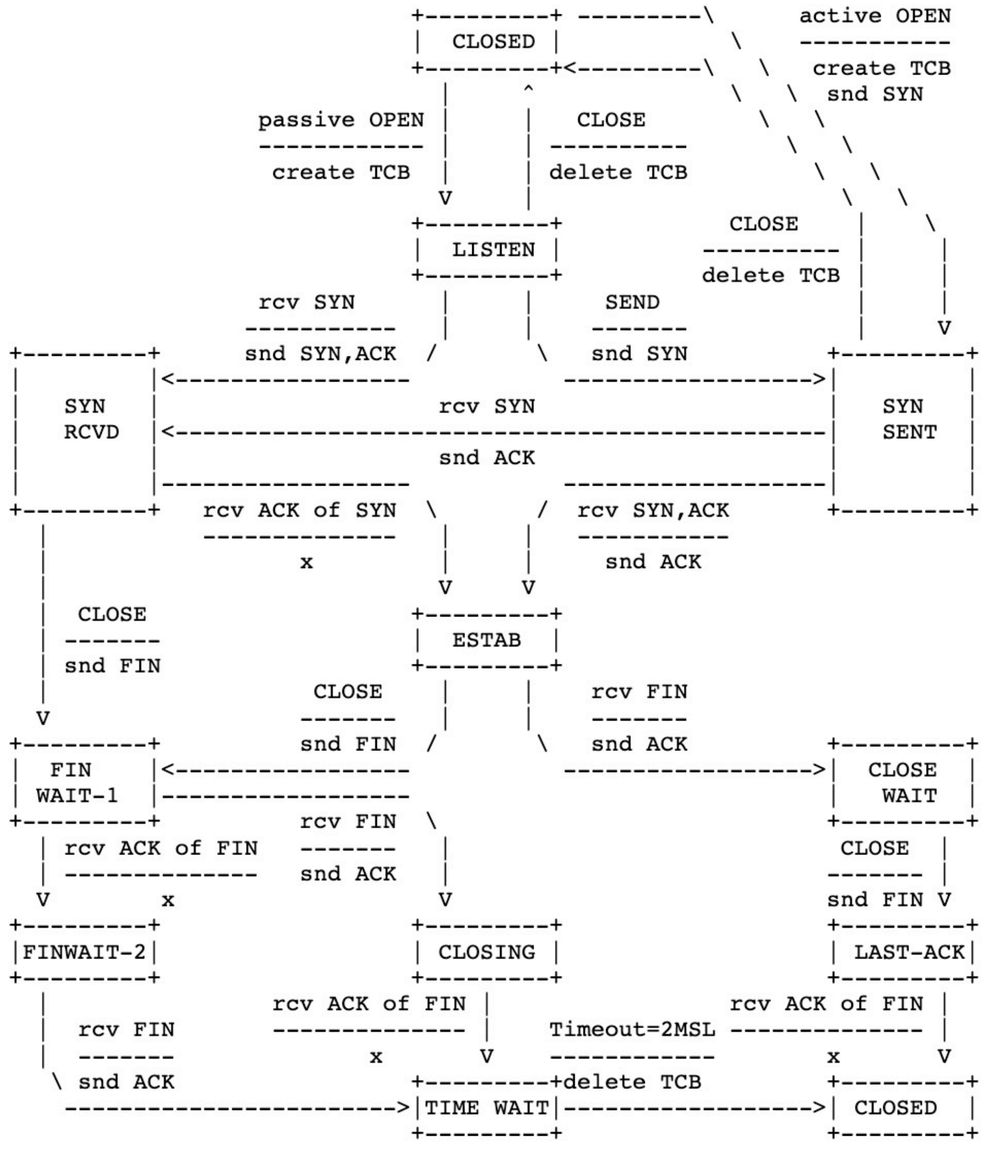
2. L3 - 白盒交换机

SONiC:



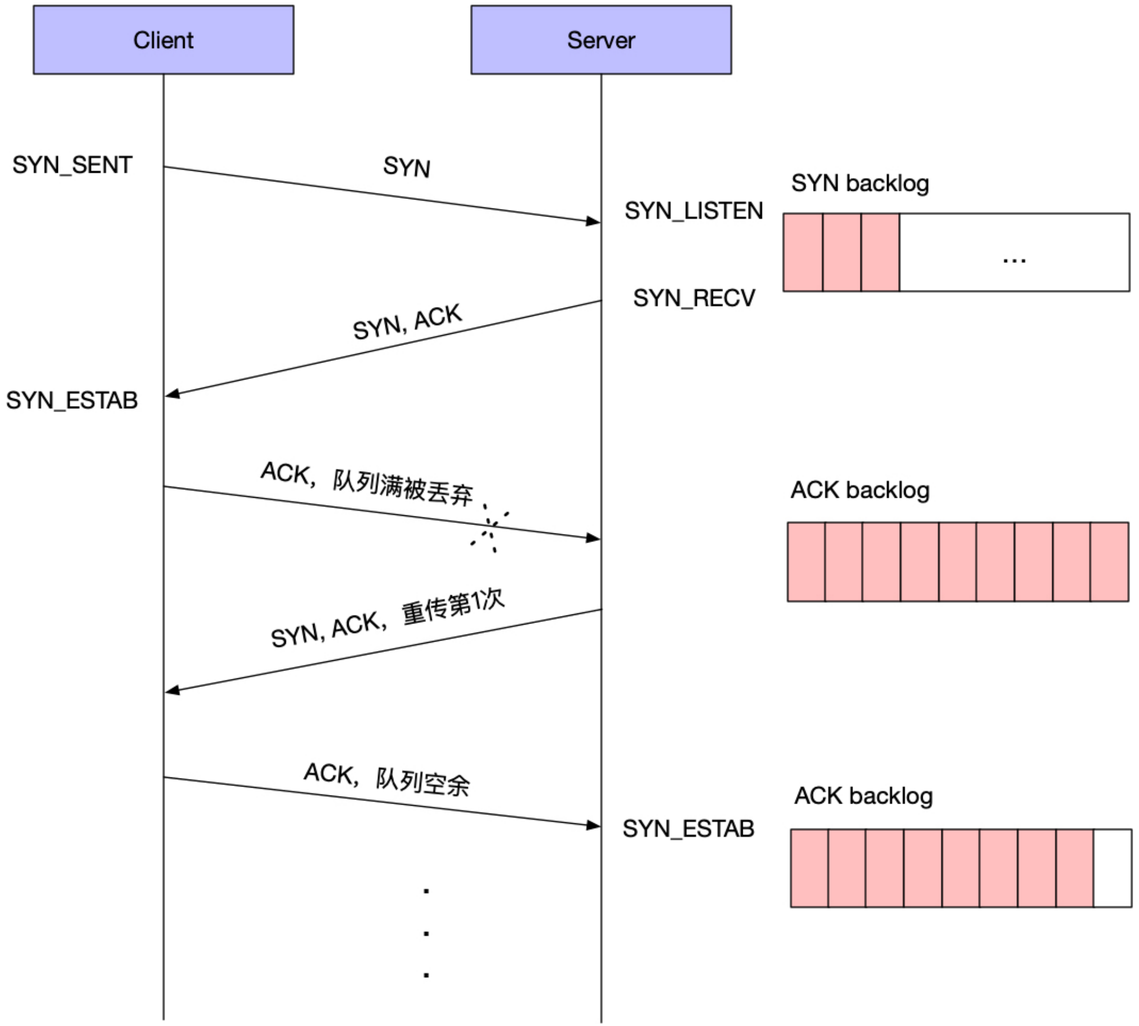
1. 硬件与软件异构
2. 基于linux内核，自定义网络能力
3. 助力SDN、vDC
4. 市场未清晰

3. L4 - TCP - 状態



TCP Connection State Diagram
Figure 6.

3. L4 - TCP - 半连接



net/core/request_sock.c

```c
/*
 * Maximum number of SYN_RECV sockets in queue per LISTEN socket.
 * One SYN_RECV socket costs about 80bytes on a 32bit machine.
 * It would be better to replace it with a global counter for all sockets
 * but then some measure against one socket starving all other sockets
 * would be needed.
 *
 * The minimum value of it is 128. Experiments with real servers show that
 * it is absolutely not enough even at 100conn/sec. 256 cures most
 * of problems.
 * This value is adjusted to 128 for low memory machines,
 * and it will increase in proportion to the memory of machine.
 * Note : Dont forget somaxconn that may limit backlog too.
 */
int sysctl_max_syn_backlog = 256;
EXPORT_SYMBOL(sysctl_max_syn_backlog);

int reqsk_queue_alloc(struct request_sock_queue *queue,
              unsigned int nr_table_entries)
{
    size_t lopt_size = sizeof(struct listen_sock);
    struct listen_sock *lopt;

    nr_table_entries = min_t(u32, nr_table_entries, sysctl_max_syn_backlog);
    nr_table_entries = max_t(u32, nr_table_entries, 8);
    nr_table_entries = roundup_pow_of_two(nr_table_entries + 1);
    lopt_size += nr_table_entries * sizeof(struct request_sock *);
    if (lopt_size > PAGE_SIZE)
        lopt = vzalloc(lopt_size);
    else
        lopt = kzalloc(lopt_size, GFP_KERNEL);
    if (lopt == NULL)
        return -ENOMEM;

    for (lopt->max_qlen_log = 3;
         (1 << lopt->max_qlen_log) < nr_table_entries;
         lopt->max_qlen_log++);

    get_random_bytes(&lopt->hash_rnd, sizeof(lopt->hash_rnd));
    rwlock_init(&queue->syn_wait_lock);
    queue->rskq_accept_head = NULL;
    lopt->nr_table_entries = nr_table_entries;

    write_lock_bh(&queue->syn_wait_lock);
    queue->listen_opt = lopt;
    write_unlock_bh(&queue->syn_wait_lock);

    return 0;
}
```

net/ipv4/af_inet.c

```c
/*
 *  Move a socket into listening state.
 */
int inet_listen(struct socket *sock, int backlog)
{
    struct sock *sk = sock->sk;
    unsigned char old_state;
    int err;

    lock_sock(sk);

    err = -EINVAL;
    if (sock->state != SS_UNCONNECTED || sock->type != SOCK_STREAM)
        goto out;

    old_state = sk->sk_state;
    if (!((1 << old_state) & (TCPF_CLOSE | TCPF_LISTEN)))
        goto out;

    /* Really, if the socket is already in listen state
     * we can only allow the backlog to be adjusted.
     */
    if (old_state != TCP_LISTEN) {
        /* Check special setups for testing purpose to enable TFO w/o
         * requiring TCP_FASTOPEN sockopt.
         * Note that only TCP sockets (SOCK_STREAM) will reach here.
         * Also fastopenq may already been allocated because this
         * socket was in TCP_LISTEN state previously but was
         * shutdown() (rather than close()).
         */
        if ((sysctl_tcp_fastopen & TFO_SERVER_ENABLE) != 0 &&
            inet_csk(sk)->icsk_accept_queue.fastopenq == NULL) {
            if ((sysctl_tcp_fastopen & TFO_SERVER_WO_SOCKOPT1) != 0)
                err = fastopen_init_queue(sk, backlog);
            else if ((sysctl_tcp_fastopen &
                  TFO_SERVER_WO_SOCKOPT2) != 0)
                err = fastopen_init_queue(sk,
                    ((uint)sysctl_tcp_fastopen) >> 16);
            else
                err = 0;
            if (err)
                goto out;
        }
        err = inet_csk_listen_start(sk, backlog);
        if (err)
            goto out;
    }
    sk->sk_max_ack_backlog = backlog;
    err = 0;

out:
    release_sock(sk);
    return err;
}
```

3. L4 - TCP - 半连接

```
> sysctl -w net.ipv4.tcp_max_syn_backlog=128
> sysctl -w net.ipv4.tcp_syncookies=0
> sysctl -w net.ipv4.tcp_syn_retries=10
> sysctl -w net.core.somaxconn=128
```

```
probe kernel.function("tcp_v4_conn_request").return {
    ret = $return
    tcphdr = __get_skb_tcphdr($skb);
    dport = __tcp_skb_dport(tcphdr);
    if (dport == 1202)
    {
        printf("reach here\n");
        syn_qlen = @cast($sk, "struct inet_connection_sock")->
            icsk_accept_queue->listen_opt->qlen;
        max_syn_qlen_log = @cast($sk, "struct inet_connection_sock")->
            icsk_accept_queue->listen_opt->max_qlen_log;
        max_syn_qlen = (1 << max_syn_qlen_log);
        printf("syn queue: syn_qlen=%d, max_syn_qlen=%d\n",
         syn_qlen, max_syn_qlen);
        printf("ack queue: ack_qlen=%d, max_ack_qlen=%d\n",
         $sk->sk_ack_backlog, $sk->sk_max_ack_backlog);
        printf("ret=%d\n", ret);
        print_backtrace();
    }
}
```

```
int main(int argc, char *argv[]) {
    int SERVER_PORT = 8877;
    struct sockaddr_in server_address;
    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(SERVER_PORT);
    server_address.sin_addr.s_addr = htonl(INADDR_ANY);
    int listen_sock;
    if ((listen_sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        printf("could not create listen socket\n");
        return 1;
    }
    if ((bind(listen_sock, (struct sockaddr *)&server_address,
              sizeof(server_address))) < 0) {
        printf("could not bind socket\n");
        return 1;
    }
    int backlog = 0;
    if (listen(listen_sock, backlog) < 0) {
        printf("could not open socket for listening\n");
        return 1;
    }

    std::map<int, int> socks;
    struct sockaddr_in client_address;
    socklen_t client_address_len = sizeof(client_address);

    while (true) {
        int sock;
        if ((sock = accept(listen_sock, (struct sockaddr *)&client_address,
                           &client_address_len)) < 0) {
            printf("could not open a socket to accept data\n");
            continue;
        }
        socks[client_address.sin_addr] = sock;
        int n = 0;
        int len = 0, maxlen = 100;
        char buffer[maxlen];
        char *pbuffer = buffer;

        printf("client connected with ip address: %s\n",
               inet_ntoa(client_address.sin_addr));

        while ((n = recv(sock, pbuffer, maxlen, 0)) > 0) {
            pbuffer += n;
            maxlen -= n;
            len += n;
            printf("received: %s", buffer);
            send(sock, buffer, len, 0);
        }
        close(sock);
    }
    close(listen_sock);
    return 0;
}
```

3. L4 - TCP - 其他话题

拥塞控制

回收优化

缓冲区配置

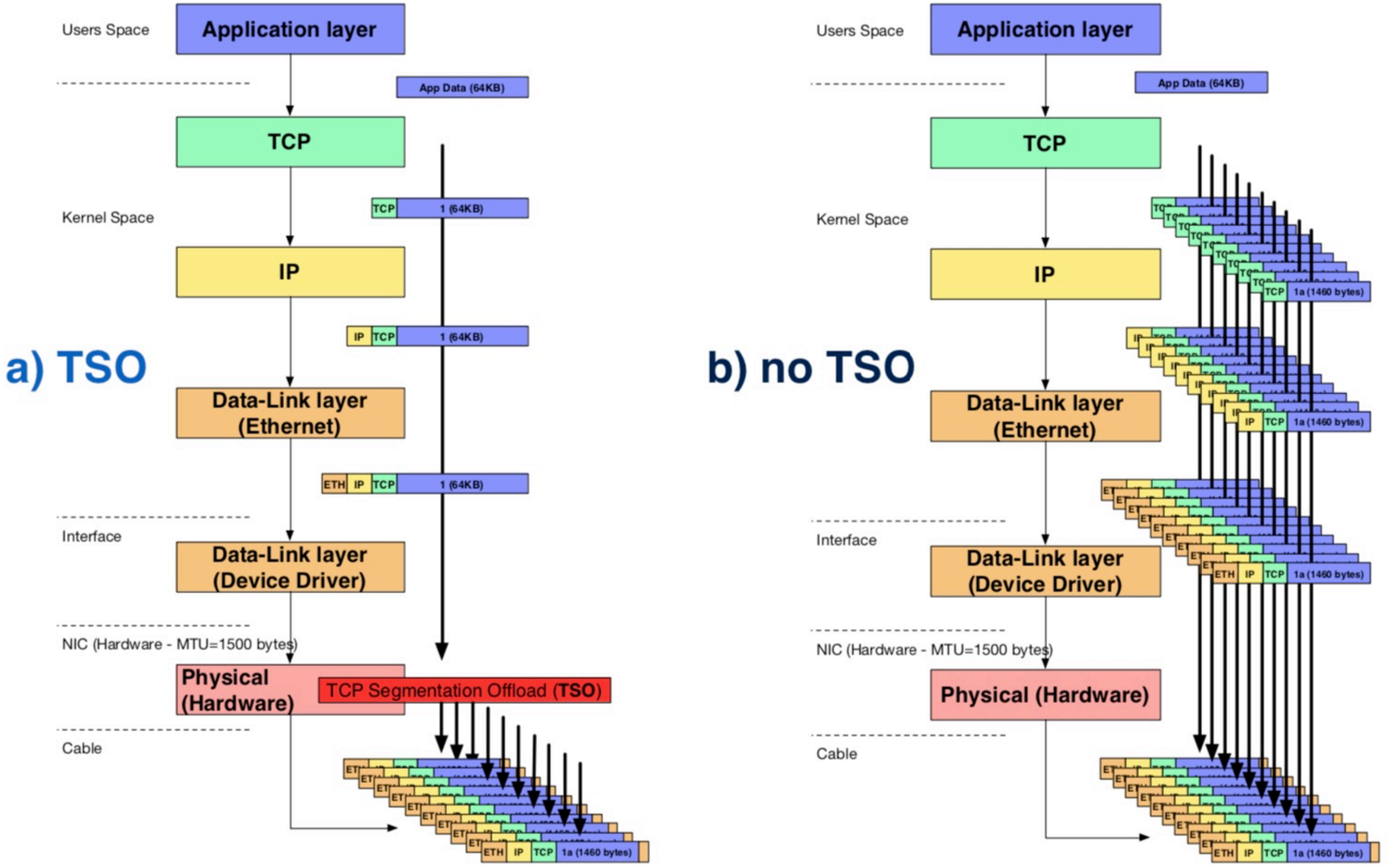
心跳配置

重传配置

等待配置

智能offload

3. L4 - TCP - TSO



“Software segmentation offloading for FreeBSD - Stefano Garzarella”

3. L4 - UDP - 阻断

RFC1122:

4.1.3.3 ICMP Messages

UDP MUST pass to the application layer all ICMP error messages that it receives from the IP layer. Conceptually at least, this may be accomplished with an upcall to the ERROR_REPORT routine (see Section 4.2.4.1).

DISCUSSION:

Note that ICMP error messages resulting from sending a UDP datagram are received asynchronously. A UDP-based application that wants to receive ICMP error messages is responsible for maintaining the state necessary to demultiplex these messages when they arrive; for example, the application may keep a pending receive operation for this purpose. The application is also responsible to avoid confusion from a delayed ICMP error message resulting from an earlier use of the same port(s).

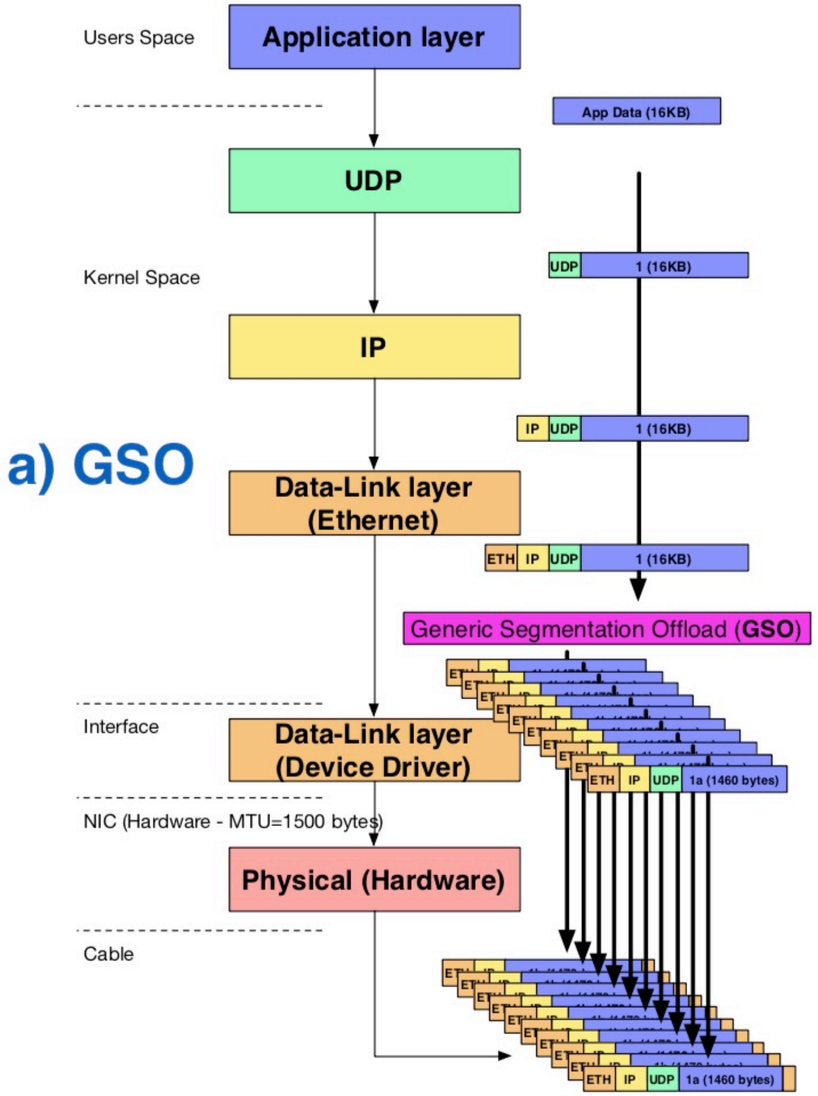
net/ipv4/udp.c:

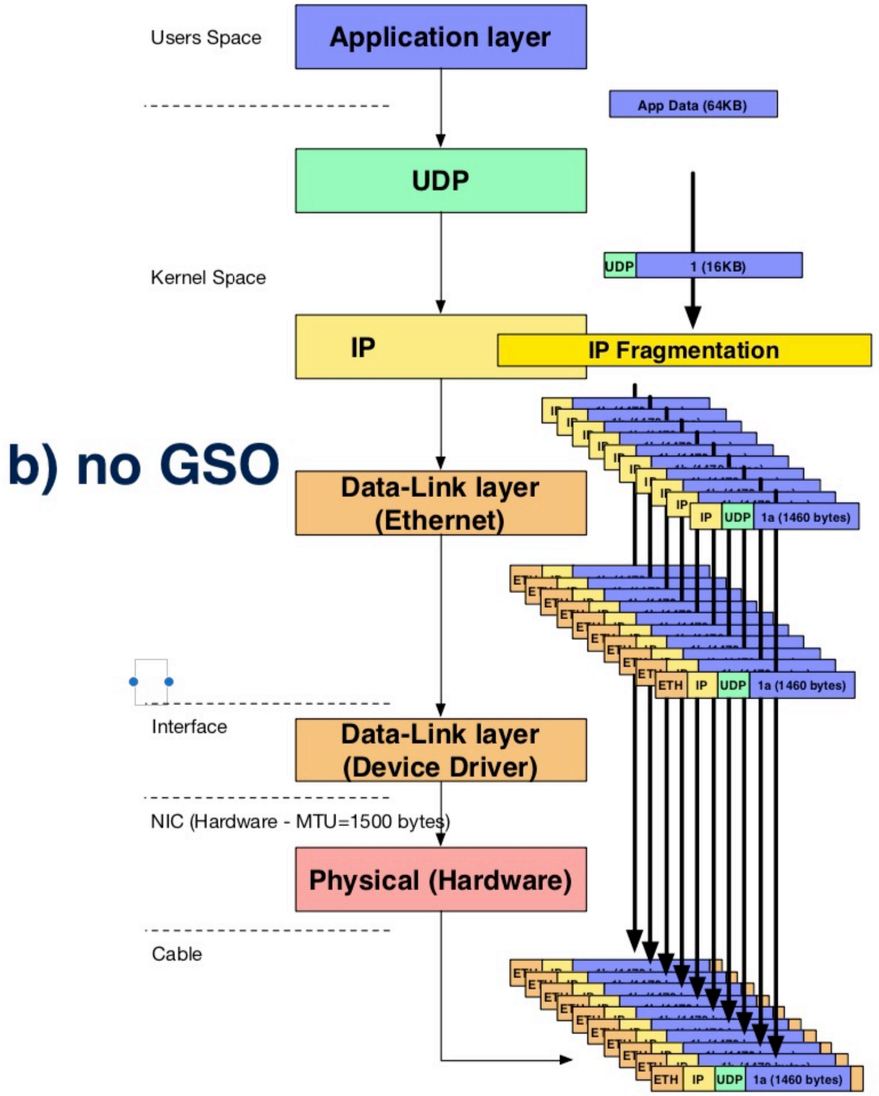
```
    /*
     *      RFC1122: OK.  Passes ICMP errors back to application, as per
     *  4.1.3.3.
     */
    if (!inet->recverr) {
        if (!harderr || sk->sk_state != TCP_ESTABLISHED)
            goto out;
    } else
        ip_icmp_error(sk, skb, err, uh->dest, info, (u8 *)(uh+1));

    sk->sk_err = err;
    sk->sk_error_report(sk);
out:
    sock_put(sk);
```

udp.stp:

```
global count=0

probe kernel.function("__udp4_lib_err") {
    count++
        if (count % 5 == 0)
            printf( "sys_sync called %d times\n", count);
}

probe timer.ms(10000){
    printf(" %d times\n\n",count);
}
```

3. L4 - UDP - GSO

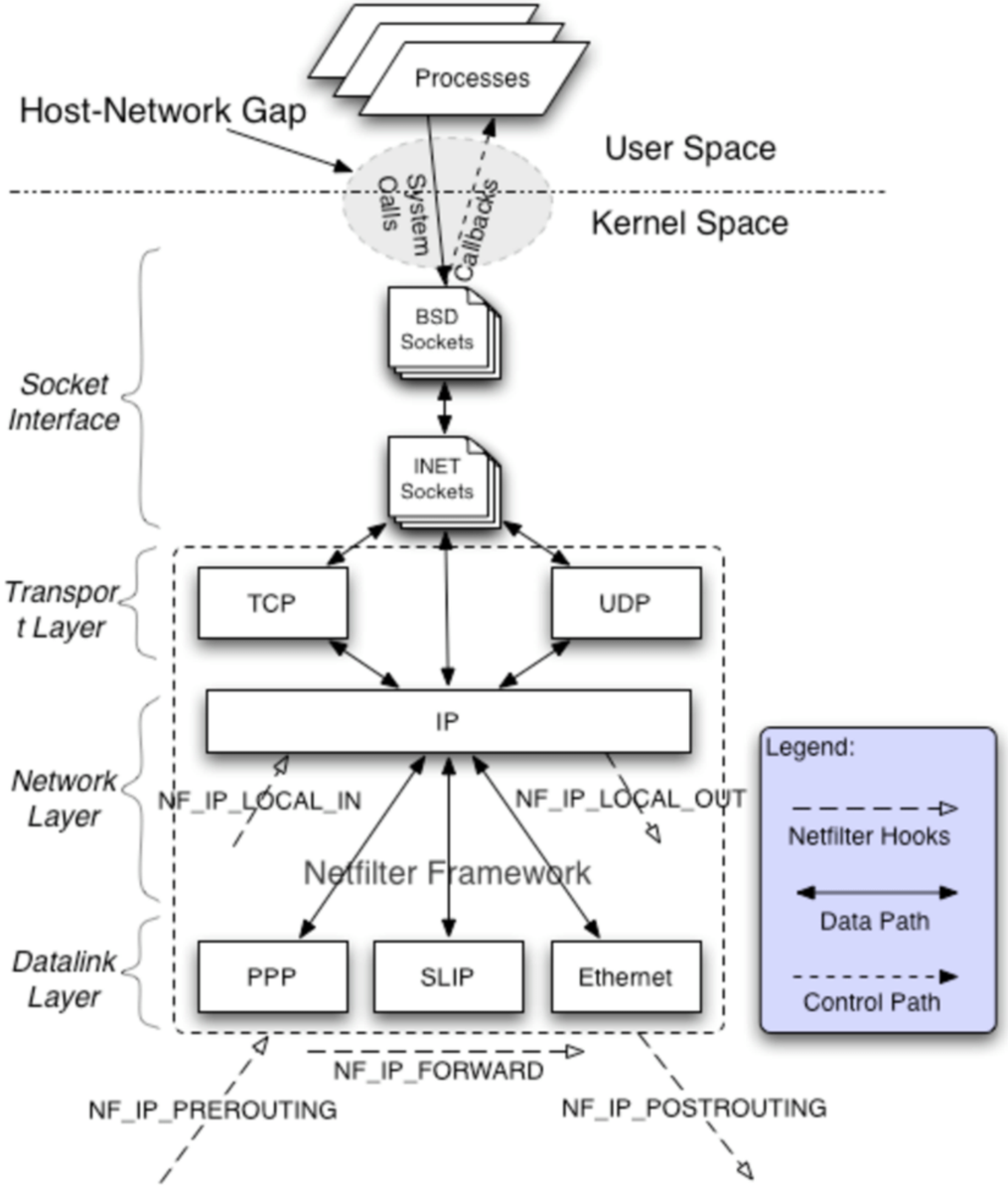

a) GSO
Users Space
Application layer
App Data (16KB)
UDP
Kernel Space
UDP 1 (16KB)
IP
IP UDP 1 (16KB)
Data-Link layer (Ethernet)
ETH IP UDP 1 (16KB)
Generic Segmentation Offload (GSO)
Interface
Data-Link layer (Device Driver)
ETH IP UDP 1a (1460 bytes)
NIC (Hardware - MTU=1500 bytes)
Physical (Hardware)
Cable


b) no GSO
Users Space
Application layer
App Data (64KB)
UDP
Kernel Space
UDP 1 (16KB)
IP
IP Fragmentation
Data-Link layer (Ethernet)
IP UDP 1a (1460 bytes)
ETH IP UDP 1a (1460 bytes)
Interface
Data-Link layer (Device Driver)
NIC (Hardware - MTU=1500 bytes)
Physical (Hardware)
Cable

4. L7 - 基础服务&应用协议

DNS	HTTP
DHCP	NFS
IPAM	SNMP
YUM	FTP
NTP	TELNET
…	…

5. Protocol Stack (bottom half)



6. ∞


IEEE


IETF


ETSI


onos
Open Network Operating System


CERTIFIED
SDN ENGINEER


openstack


OPEN
DAYLIGHT


OPNFV


kubernetes

References

1. https://www.kernel.org/doc/html/latest/networking/index.html
2. https://ieeexplore.ieee.org/document/8457469
3. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/dc
4. https://ceph.io/geen-categorie/ceph-loves-jumbo-frames/
5. https://fast.dpdk.org/doc/perf/DPDK_18_02_Intel_NIC_performance_report.pdf
6. https://gitlab.com/wireshark/wireshark/raw/master/manuf
7. https://en.wikipedia.org/wiki/MAC_address
8. https://cora.ucc.ie/bitstream/handle/10468/3365/1744.pdf?sequence=1
9. https://tools.ietf.org/html/rfc826
10. https://tools.ietf.org/html/rfc791
11. http://linux-ip.net/linux-ip/linux-ip-single.html#routing-selection
12. https://man7.org/linux/man-pages/man7/netlink.7.html
13. https://docs.projectcalico.org/networking/bgp
14. https://tools.ietf.org/html/rfc1122#page-78