linux libpcap的性能问题，请大家注意绕行。

内核代码中，ip_rcv是ip层收包的主入口函数，该函数由软中断调用。存放数据包的sk_buff结构包含有目的地ip和端口信息，此时ip层进行检查，如果目的地ip不是本机，且没有开启转发的话，则将包丢弃，如果配置了netfilter，则按照配置规则对包进行转发。

tcp_v4_rcv是tcp层收包的接收入口，其调用__inet_lookup_skb函数查到数据包需要往哪个socket传送，之后将数据包放入tcp层收包队列中，如果应用层有read之类的函数调用，队列中的包将被取出。

最近遇到一个问题，就是libpcap的收包，比tcpdump的收包，要慢。

然后修改测试代码如下：

```
#include <pcap.h>               /*libpcap*/

static pcap_t * pcap_http_in;

int initPcapIn_http()
{
      int snaplen = 1518;//以太网数据包，最大长度为1518bytes
      int promisc = 1;//混杂模式
      int timeout = 1000;
      char errbuf[PCAP_ERRBUF_SIZE];//内核缓冲区大小
    /*这个设备号需要根据测试服务器更改*/
    char * _pcap_in = "enp8s0f1";
    char * _bpf_filter = "tcp[13]=24";

    /*打开输入设备或者文件*/
    if((pcap_http_in = pcap_open_live(_pcap_in, snaplen, promisc, timeout, errbuf)) == NULL)
    {
        printf("pcap_open_live(%s) error, %s\n", _pcap_in, errbuf);
        pcap_http_in = pcap_open_offline(_pcap_in, errbuf);
        if(pcap_http_in == NULL) {
              printf("pcap_open_offline(%s): %s\n", _pcap_in, errbuf);
        } else
            printf("Reading packets from pcap file %s...\n", _pcap_in);
    }
    else
    {
        printf("Capturing live traffic from device %s...\n", _pcap_in);

        /*设置bpf过滤参数。*/
        if(_bpf_filter!= NULL)
        {
            struct bpf_program fcode;

            if(pcap_compile(pcap_http_in, &fcode, _bpf_filter, 1, 0xFFFFFF00) < 0)
            {
                printf("pcap_compile error: '%s'\n", pcap_geterr(pcap_http_in));
            }
            else
            {
                if(pcap_setfilter(pcap_http_in, &fcode) < 0)
                {
                    printf("pcap_setfilter error: '%s'\n", pcap_geterr(pcap_http_in));
                }
```

```
                else
                    printf("Succesfully set BPF filter to '%s'\n", _bpf_filter);
            }
        }

        /*设置一些参数*/
        if(pcap_setdirection(pcap_http_in, PCAP_D_IN)<0)     /*只抓入向包*/
        {
            printf("pcap_setdirection error: '%s'\n", pcap_geterr(pcap_http_in));
        }
        else
            printf("Succesfully set direction to '%s'\n", "PCAP_D_IN");
    }

    return 0;
}

static inline unsigned long long rp_get_us(void)
{
    struct timeval tv = {0};
    gettimeofday(&tv, NULL);
    return (unsigned long long)(tv.tv_sec*1000000L + tv.tv_usec);
}

int main(int argc, char *argv[])
{
    initPcapIn_http();

    unsigned char * pkt_data = NULL;
    struct pcap_pkthdr pcap_hdr;
    struct pcap_pkthdr * pkt_hdr = &pcap_hdr;
    while(1)
    {
        while( (pkt_data = (unsigned char * )pcap_next( pcap_http_in, &pcap_hdr))!=NULL)
        {
            if(pkt_hdr->caplen == 454)
            {
                unsigned long long time1 = rp_get_us();
                printf("---BEGIN: %ld us\n",time1);
            }
        }
    }

    if (pcap_http_in)
    {
        pcap_close(pcap_http_in);
    }

    return 0;
}
```

一开始静态编译，gcc静态编译报错，/usr/bin/ld: cannot find -lc

Makefile中肯定有-static选项。这其实是静态链接时没有找到libc.a。

其实需要安装glibc-static.xxx.rpm，如glibc-static-2.12-1.107.el6_4.2.i686.rpm，或是yum install glibc-static，我最终下载的是:glibc-static-2.17-157.el7.x86_64.rpm.

结果测试发现，我们打印的---BEGIN的时间，比tcpdump对应的时间晚1ms左右，也就是1000us左右。

然后我们根据tcpdump的调用方式，发现

```
 ldd /sbin/tcpdump  |grep -i pcap
    libpcap.so.1 => /usr/local/lib/libpcap.so.1 (0x00007fd1903f1000)

ls -alrt /usr/local/lib/libpcap.so.1
lrwxrwxrwx. 1 root root 16 7月 25 19:36 /usr/local/lib/libpcap.so.1 -> libpcap.so.1.5.3
```

然后我将我的编译方式改成动态链接方式，即

```
gcc  -lpcap -g -o pcap.o pcap.c
```

发现效果很好，跟tcpdump差不多，也就是说，动态链接的lpcap的性能比静态链接的lpcap的性能要好。颠覆了我的认知，因为我一直认为静态链接快一点是有可能的。

我下载的源码是http://www.tcpdump.org/release/官网的，系统自带的版本和我的版本号一致。

发现不管是gcc -O2还是O3都是如此，我的静态链接的库就是慢，然后将tcpdump官网的libpcap库改成动态链接，还是慢。对比如下：

```
自己tcpdump官网下载的1.5.3的libpcap如下
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.103021>
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.095322>
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.101384>
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.100031>

对应的系统自带的1.5.3版本如下：
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.000139>
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.000061>
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.000231>
poll([{fd=3, events=POLLIN}], 1, 1000)  = 1 ([{fd=3, revents=POLLIN}]) <0.000062>
```

从参数看，是一模一样的，但是调用的消耗看，前者明显慢，直觉告诉我，应该看fd的属性，所以针对属性又单独跟踪了一次：

```
系统自带的1.5.3版本：
[root@localhost libpcap-1.5.3]# strace -e setsockopt ./pcaptest
setsockopt(3, SOL_PACKET, PACKET_ADD_MEMBERSHIP, "\3\0\0\0\1\0\0\0\0\0\0\0\0\0\0\0", 16) = 0
setsockopt(3, SOL_PACKET, PACKET_AUXDATA, [1], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_VERSION, [1], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_RESERVE, [4], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_RX_RING, {block_size=4096, block_nr=655, frame_size=1600, frame
Capturing live traffic from device enp5s0...
setsockopt(3, SOL_SOCKET, SO_ATTACH_FILTER, "\1\0\0\0\0\0\0\224\246c\0\0\0\0\0", 16) = 0
setsockopt(3, SOL_SOCKET, SO_ATTACH_FILTER, "\v\0\0\0\0\0\0\260\276\357\1\0\0\0\0", 16) = 0
我在tcpdump官网下载的libpcap版本：
[root@localhost libpcap-1.5.3]# strace -e setsockopt ./pcaptest
setsockopt(3, SOL_PACKET, PACKET_ADD_MEMBERSHIP, "\3\0\0\0\1\0\0\0\0\0\0\0\0\0\0\0", 16) = 0
```

```
setsockopt(3, SOL_PACKET, PACKET_AUXDATA, [1], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_VERSION, [2], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_RESERVE, [4], 4) = 0
setsockopt(3, SOL_PACKET, PACKET_RX_RING, "\0\0\2\0\20\0\0\0\0\2\0\20\0\0\0\350\3\0\0\0\0\0\
Capturing live traffic from device enp5s0...
setsockopt(3, SOL_SOCKET, SO_ATTACH_FILTER, "\1\0\0\0\0\0\0\224\246c\0\0\0\0\0", 16) = 0
setsockopt(3, SOL_SOCKET, SO_ATTACH_FILTER, "\v\0\0\0\0\0\0P\6\343\1\0\0\0\0", 16) = 0
Succesfully set BPF filter to 'tcp[13]=24'
Succesfully set direction to 'PCAP_D_IN'
```

果然参数不一样，设置的PACKET_VERSION，快的是2，慢的是3.

同样的版本号，难道代码有区别，走查代码流程，再加上gdb和strace，确定是PACKET_VERSION的问题。

```
(gdb) p *(struct pcap_linux*)handle->priv
$2 = {packets_read = 26, proc_dropped = 76591622, stat = {ps_recv = 0, ps_drop = 0, ps_ifdrop =
  filter_in_userland = 0, blocks_to_filter_in_userland = 0, must_do_on_close = 0, timeout = 1000
  lo_ifindex = 1, oldmode = 0, mondevice = 0x0, mmapbuf = 0x7ffff513d000 "\002", mmapbuflen = 20
  tp_hdrlen = 36, oneshot_buffer = 0x81c940 "", current_packet = 0x0, packets_left = 0}
```

PACKET_VERSION就是一个宏决定的，后来发现，系统自带的版本，没有定义define HAVE_TPACKET3，所以不会走V3的版本。

那么，下一步就需要排查，怎么会慢，慢在哪里。

用户态没有问题，直接调用gettimeofday打印下时间。内核态要用stap跟踪了。

由于是抓包，那么在哪里下桩呢?

我们先来看pacap的版本，理清楚调用。

在 pcap_activate_linux 函数中，会先设置handle的默认的一些回调。

```
    handle->inject_op = pcap_inject_linux;
    handle->setfilter_op = pcap_setfilter_linux;
    handle->setdirection_op = pcap_setdirection_linux;
    handle->set_datalink_op = pcap_set_datalink_linux;
    handle->getnonblock_op = pcap_getnonblock_fd;
    handle->setnonblock_op = pcap_setnonblock_fd;
    handle->cleanup_op = pcap_cleanup_linux;
    handle->read_op = pcap_read_linux;
    handle->stats_op = pcap_stats_linux;
```

在此之后，会先尝试activate_new方法，如果失败的话，则会回退到activate_old方法，其实这种命名不太好，因为后续内核发展了，总不能叫activate_new_new.

在activate_new中，

```
sock_fd = is_any_device ?
socket(PF_PACKET, SOCK_DGRAM, htons(ETH_P_ALL))
socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));:后面要用到，协议类型和socket类型
```

这个申请socket单独拿出来说，是因为根据是否抓包带-any 参数，后面下钩子的地方不一样。

如果activate_new调用成功，则会继续判断activate_mmap ，这个函数其实就是测试内核是否支持mmap的方式来抓取报文，如果支持的话，就使用mmap的方式来获取报文，否则，就会回退到

之前的方法使用。针对mmap收包的情况，tp_version也有三个version，分别是v1，v2，v3，代码中会尝试先设置v3，如果不行则设置v2，以此类推。最终会调用 init_tpacket 来设置socket属性。先通过 getsockopt(handle->fd, SOL_PACKET, PACKET_HDRLEN, &val, &len) 来获取能力，然后使用

setsockopt(handle->fd, SOL_PACKET, PACKET_VERSION, &val,sizeof(val)) 来设置。PACKET_MMAP非常高效，它提供一个映射到用户空间的大小可配置的环形缓冲区。这种方式，读取报文只需要等待报文就可以了，大部分情况下不需要系统调用（其实poll也是一次系统调用）。通过内核空间和用户空间共享的缓冲区还可以起到减少数据拷贝的作用。
当然为了提高捕获的性能，不仅仅只是PACKET_MMAP。如果你在捕获一个高速网络中的数据，你应该检查NIC是否支持一些中断负载缓和机制或者是NAPI，确定开启这些措施。
PACKET_MMAP减少了系统调用，不用recvmsg就可以读取到捕获的报文，相比原始套接字+recvfrom的方式，减少了一次拷贝和一次系统调用，但是低版本的libpcap是不支持PACKET_MMAP，比如libpcap 0.8.1以及之前的版本都不支持，具体的资料，大家可以参考Document/networking/packet_mmap.txt，**而本文最终的问题，刚好出在PACKET_MMAP上面，所谓成也萧何，败也萧何，下面继续分析代码：**

activate_mmap -->|prepare_tpacket_socket-->init_tpacket

-->|create_ring（关于ring的一些设置），很关键地调用了**if (setsockopt(handle->fd, SOL_PACKET, PACKET_RX_RING,(void *) &req, sizeof(req)))**

-->|pcap_read_linux_mmap_v3（1,2），以及handle的其他一些回调设置。

继续libpcap的代码：

```
    switch (handlep->tp_version) {
    case TPACKET_V1:
        handle->read_op = pcap_read_linux_mmap_v1;
        break;
#ifdef HAVE_TPACKET2
    case TPACKET_V2:
        handle->read_op = pcap_read_linux_mmap_v2;
        break;
#endif
#ifdef HAVE_TPACKET3
    case TPACKET_V3:
        handle->read_op = pcap_read_linux_mmap_v3;
        break;
#endif
    }
    handle->cleanup_op = pcap_cleanup_linux_mmap;
    handle->setfilter_op = pcap_setfilter_linux_mmap;
    handle->setnonblock_op = pcap_setnonblock_mmap;
    handle->getnonblock_op = pcap_getnonblock_mmap;
    handle->oneshot_callback = pcap_oneshot_mmap;
    handle->selectable_fd = handle->fd;
```

如果activate_new失败，则会尝试activate_old，那么创建socket就会使用：

```
handle->fd = socket(PF_INET, SOCK_PACKET, htons(ETH_P_ALL));
```

这个注意和上面activate_new使用的socket方法相区别。这种模式不支持-any参数。

因为我目前使用的内核是3.10.0+，所以不会走active_old流程。

在libpcap的库中，pcap_open_live-->|pcap_create--->pcap_create_interface,设置handle->activate_op = pcap_activate_linux;

|pcap_activate-->pcap_activate_linux

在tcpdump的代码中，直接就是main函数调用pcap_create和pcap_activate。然后在pcap_loop中循环收包，调用链分析结束。

下一步，需要了解，在内核态收包和在用户态收包的钩子下在哪里比较合适。因为使用的socket是SOCK_RAW，family是PF_PACKET，而且我们使用的是mmap收包，所以有必要看一下

PF_PACKET针对mmap的收包函数。所以就针对SOCK_RAW的收包去下钩子。

从创建socket的地方看起：

```
static const struct net_proto_family packet_family_ops = {
    .family =    PF_PACKET,
    .create =    packet_create,
    .owner    =    THIS_MODULE,
};

static int packet_create(struct net *net, struct socket *sock, int protocol,
             int kern)
{
。。。
    spin_lock_init(&po->bind_lock);
    mutex_init(&po->pg_vec_lock);
    po->prot_hook.func = packet_rcv;
。。。
}
```

那么看起来要在packet_rcv 下钩子，但是因为libpcap里面针对create_ring的函数调用**if (setsockopt(handle->fd, SOL_PACKET, PACKET_RX_RING,(void *) &req, sizeof(req)))**

使得packet_setsockopt函数中针对po->prot_hook.func 修改为了 tpacket_rcv ，所以我们应该在这个函数下钩子。tpacket_rcv是PACKET_MMAP的实现，packet_rcv是普通AF_PACKET的实现。

```
static int
packet_setsockopt(struct socket *sock, int level, int optname, char __user *optval, unsigned int
{
。。。。。
        case PACKET_RX_RING:
    case PACKET_TX_RING:
    {
        union tpacket_req_u req_u;
        int len;

        switch (po->tp_version) {
        case TPACKET_V1:
        case TPACKET_V2:
```

```
            len = sizeof(req_u.req);
            break;
        case TPACKET_V3:
        default:
            len = sizeof(req_u.req3);
            break;
        }
        if (optlen < len)
            return -EINVAL;
        if (pkt_sk(sk)->has_vnet_hdr)
            return -EINVAL;
        if (copy_from_user(&req_u.req, optval, len))
            return -EFAULT;
        return packet_set_ring(sk, &req_u, 0,
            optname == PACKET_TX_RING);
。。。。。。
    }


static int packet_set_ring(struct sock *sk, union tpacket_req_u *req_u,
        int closing, int tx_ring)
{
。。。。
    po->prot_hook.func = (po->rx_ring.pg_vec) ?
                        tpacket_rcv : packet_rcv;-------------决定了下桩的函数，我们是tpacket_rcv

。。。。。
```

这个偷懒一下，借用同事涂强的脚本，

```
probe kernel.function("tpacket_rcv") {
        iphdr = __get_skb_iphdr($skb)
        saddr = format_ipaddr(__ip_skb_saddr(iphdr), %{ /* pure */ AF_INET %})
        daddr = format_ipaddr(__ip_skb_daddr(iphdr), %{ /* pure */ AF_INET %})

        tcphdr = __get_skb_tcphdr($skb)
        dport = __tcp_skb_dport(tcphdr)
        sport = __tcp_skb_sport(tcphdr)
        psh = __tcp_skb_psh(tcphdr)
        ack = __tcp_skb_ack(tcphdr)

        if(dport == 80 && psh == 1 && ack == 1) {
                printf("%-25s %-10d, ts:%ld %s %s %d %d\n", execname(), pid(), gettimeofday_us()
        }
}
```

取样数据如下：

```
swapper/2 0 , ts:1516709467430812 10.74.44.16 10.75.9.158 80 53217 //内核时间戳
swapper/2 0 , ts:1516709467430822 10.74.44.16 10.75.9.158 80 53217

---BEGIN: 1516709467431852 us //tcpdump官网下载的libpcap版本
```

```
---BEGIN: 1516709467430831 us //centos官网下载的系统自带libpcap版本
```

可以看出，tcpdump官网下载的版本，也就是TPACKET_V3使能的，在内核收到包的时候，还是和TPACKET_V2的差10us，但是到用户态poll收包，则慢了1000us左右。说明redhat系列内核对

TPACKET_V3支持得肯定有问题。

rpm -qpi libpcap --changelog 查询变更记录，找到了相关信息：

* Tue Dec 02 2014 Michal Sekletar <msekleta@redhat.com> - 14:1.5.3-4
- disable TPACKET_V3 memory mapped packet capture on AF_PACKET socket, use TPACKET_V2 instead (#1085096)

在 https://git.centos.org/blobdiff/rpms!libpcap.git/ed18a5631cc5de8fa95805d0cfd29a0678ea1458/SPECS!libpcap.spec中，找到了对应的patch。

Patch5: 0001-pcap-linux-don-t-use-TPACKETV3-for-memory-mmapped-ca.patch

uname -a
Linux centos7 3.10.0+

综上所述，如果是redhat系列，需要关闭TPACKETV3，不能直接使用tcpdump官网的libpcap包，suse的不存在这个问题。

水平有限，如果有错误，请帮忙提醒我。如果您觉得本文对您有帮助，可以点击下面的 推荐 支持一下我。