

# 基于套接字和 LKM 的网络设备接口编程实现

许言午

(电子科技大学通信抗干扰技术国家级重点实验室, 四川 成都 611731)

**摘要:** 在 Linux 操作系统上自定义虚拟的网络接口, 对于通信协议验证、网络设计实验有着重要作用。本文对一种 VNI 的编程实现进行了介绍。VNI 的发送部分基于 raw socket 实现; 接收部分以注册内核 LKM 的形式实现。并且进行了软件运行测试、抓包对比和结果分析。

**关键词:** 虚拟网络接口; 套接字编程; Linux 内核

## 1 引言

在网络协议设计的相关课题中, 尤其是需要搭建半实物仿真验证平台时, 为了验证新设计的上层协议(如组网、路由等网络层及之上的功能)的正确性和有效性, 经常有这样的需求: 改变内核原有的协议模块, 或是(仿真的协议实现在应用层时)让用户空间越过内核中的传输层、网络层, 而直接与网口交互数据。本实验在 Linux 操作系统中, 实现了一个虚拟网络接口(Virtual Network Interface, VNI)模块, 即实现了网络层协议(IP)与 MAC 协议(Ethernet)间加入自定义的协议头的效果。

本文结构安排如下: 第二节介绍设计原理, 围绕套接字和可加载内核模块; 第三节介绍 VNI 模块的设计方案; 第四节简述软件的编程实现; 第五节设计了包括 ping 报文在内的模块测试并整理了结果, 且用抓包软件 Wireshark 进行了验证; 最后总结本实验。

## 2 设计原理

本实现主要使用了两种工具, 套接字(socket)和可加载内核模块(Loadable Kernel Modules, LKM), 其中套接字又主要使用了原始套接字(raw socket)。下面分别介绍两者的基本原理以及在本实验中的应用。

### 2.1 socket

socket 是操作系统内核中的一个数据结构, 也是一种特殊的 I/O 接口, 它是网络中的节点进行进程间通信的门户。常用一个半相关描述{协议、本地地址、本地端口}来表示一个 socket; 而完整的 socket 则用一个完整相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。

socket 类型主要有三种, 流式 socket (SOCK\_STREAM)、数据报 socket (SOCK\_DGRAM) 和原始 socket (SOCK\_RAW, 或 raw socket)<sup>[1]</sup>。流式 socket 提供可靠的、面向连接的通信流, 使用 TCP 协议, 从而保证了数据传输的正确性和顺序性。数据报 socket 定义了一种无连接的服务, 使用数据报协议 UDP, 数据通过相互独立的报文进行传输, 是无序的, 并且是不可靠的。Raw socket 常用于新的网络协议实现的测试等, raw 套接字允许对底层协议如 IP 或 ICMP 进行直接访问, 甚至在链路层收发数据帧, 它功能强大, 主要用于一些自定义协议的开发。

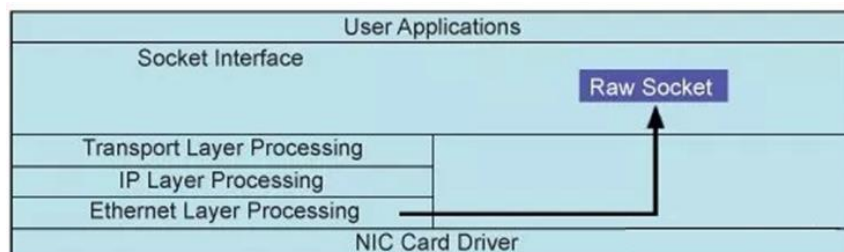


图 2-1 raw socket 与内核协议栈的关系<sup>[2]</sup>

协议栈的 raw socket 从实现上又可以分为“链路层 raw socket”和“网络层 raw socket”两大

类。其中，链路层 **raw socket** 可以直接用于接收和发送链路层的 **MAC** 帧，在发送时需要由调用者自行构造和封装 **MAC** 首部。而网络层 **raw socket** 可以直接用于接收和发送 **IP** 层的报文数据，在发送时需要自行构造 **IP** 报文头。

本实验发送部分采用的是链路层 `raw socket`，因为依靠其功能可以在应用层发送自定义的任意 MAC 帧，有极高的自由度，除基本的实现在 IP 之下 MAC 之上封装自己的 VNI 头部外，能够满足比实验目标更高的需求。`raw socket` 的实现实际上取决于操作系统<sup>[3]</sup>，在本实验中的应用将会在方案设计和软件实现中具体分析。

## 2.2 LKM

LKM 是 Linux 内核为了扩展其功能所使用的可加载内核模块[4]。LKM 的优点: 动态加载, 无须重新实现整个内核。基于此特性, LKM 常被用作特殊设备的驱动程序 (或文件系统), 如声卡的驱动程序等等。

与 raw socket 在用户空间的应用不同, LKM 在编译、加载成功后, 运行在内核空间, 与其他先载内核功能是一致的, 同样可以使用内核资源, 获取内核中的某些数据、属性信息。由于 LKM 具有这样的特性, 本实验 VNI 的接收部分以 LKM 的形式实现。

编写自定义的内核模块来处理收包，需要结合内核协议栈的功能来设计。图 2-2 概括了 Linux 协议栈网络层及以下部分的数据收发过程，以数据流经的处理函数为脉络。在非桥接模式下，网卡收到的 MAC 帧向上送至网络层的第一个应用程序接口（Application Programming Interface, API）为 `packet_type`（图 2-2 中绿色方框内），准确地来说是 `type` 对应的 `list_head` 结构。

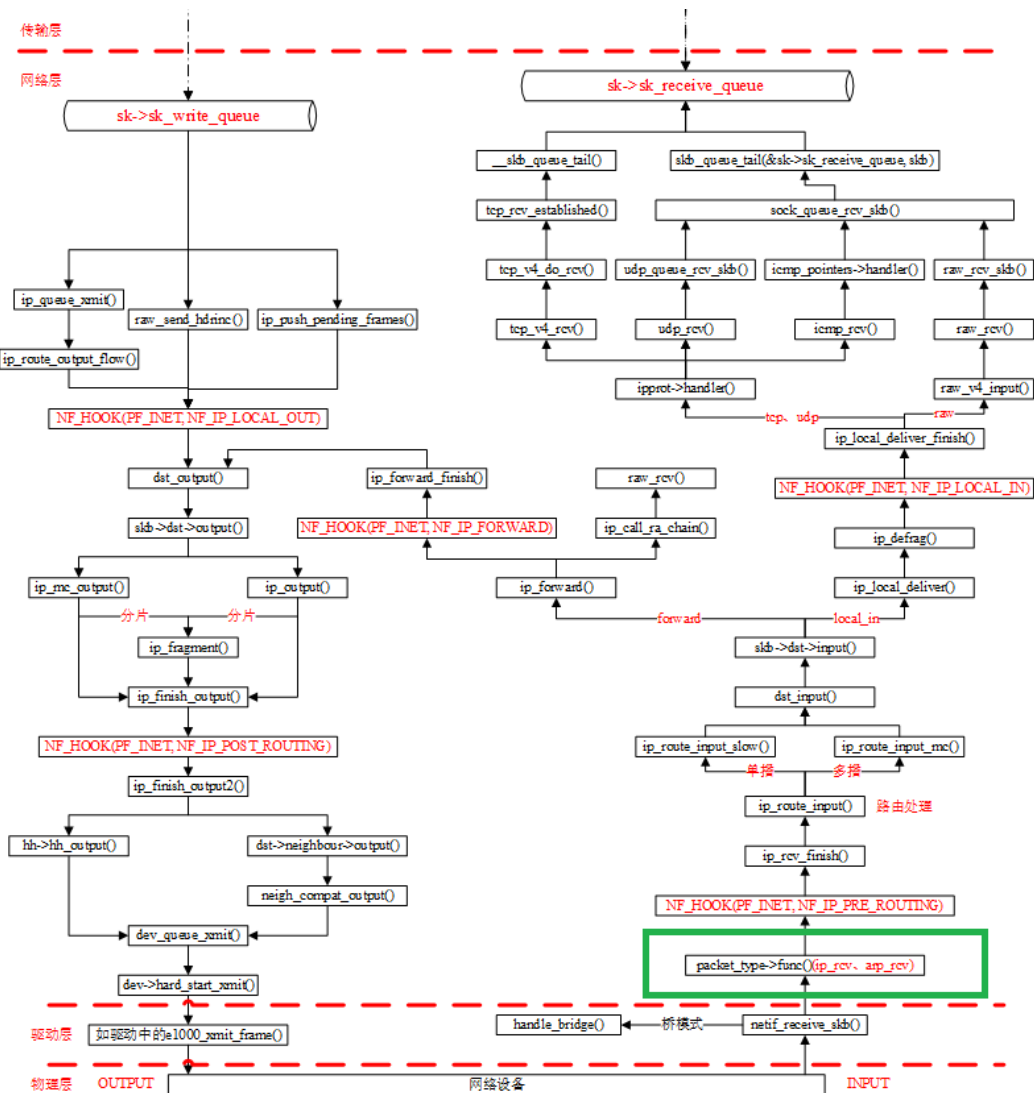


图 2-2 Linux 协议栈网络层及以下部分

内核中报文以 `sk_buff` 结构体的形式被承载，报文的传输是用 `sk_buff` 结构体的传输来实现的，而结构体在各层协议之间传输不是额外消耗资源的拷贝过程，而是通过增加协议头和移动指针来操作的[5]。内核中有很多用来处理 `sk_buff` 结构体的函数，以辅助通信协议的实现。自定义的内核模块应当在 `packet_type` 处发挥功能，即去除 VNI 头部、向上递交等。更具体的设计细节将在第三节进行分析。

### 3 设计方案

本节介绍 VNI 的设计方案，分为基于 `raw socket` 的发送部分和基于 LKM 的接收部分依次分析。

#### 3.1 基于raw socket的发送部分

题目中的 VNI 协议头实际上改变了 MAC 层和 IP 层两者的协议头，即 MAC 头部 14 字节中的 2 字节“协议类型”部分、IP 协议头部前的 6 字节自定义部分。我们不妨直接分析一个比题目更普适的需求，即发送一个除以太网帧格式（头部 14 字节长度）外完全自定义的、内容任意的以太网帧。

由第二节设计原理可知，借助链路层 `raw socket` 可以在应用层发送自定义的任意 MAC 帧。实现方法包含 5 个基本步骤（其中具体的数据、属性等的选取将在第四节软件实现中给出）：

①选择所需的协议编号。这里的“协议编号”是 `raw socket` 建立所需的一个属性，与承载的网络层协议有关，本实验中可认为取 IP 协议。

②创建 `raw socket`。主要围绕 `socket(int __domain, int __type, int __protocol)` 创建函数，其中 `__domain` 表示作用范围，应选取为链路层，`__type` 应选取为原始套接字，`__protocol` 即为已选定的协议编号。另外，还需要 `ifreq` 结构辅助，用来配置 IP 地址，激活接口，配置 MTU 等接口信息。

③设置地址类，绑定创建的 `socket`。地址类可以采用 `sockaddr_ll`，一种设备无关的物理层地址结构来辅助设计，选取地址族、上层的协议类型、接口类型、分组类型及 MAC 层地址长度（本实验为以太网帧，固定 6 字节）等。使用 `bind()` 绑定函数将 `socket` 绑定到合适的接口。

④构造数据帧。即对 VNI 分组数据自定义填充功能的实现，需要注意以太网帧长度应符合范围，MAC 帧头 14 字节结构不能破坏（因为底层仍使用以太网卡，需要符合以太网帧结构），除此之外其他的数据都是可修改的，甚至是源、目 MAC 地址。回归到题目要求，则应将 6 字节目的 MAC 地址取广播地址，2 字节类型字段改为题设值 `0xf4f0`，6 字节 VNI 头部包括 4 字节学号后四位和 2 字节帧序号，IP 报文部分为 IP 层下传的原始 IP 报文，如 84 字节的 ping 报文。

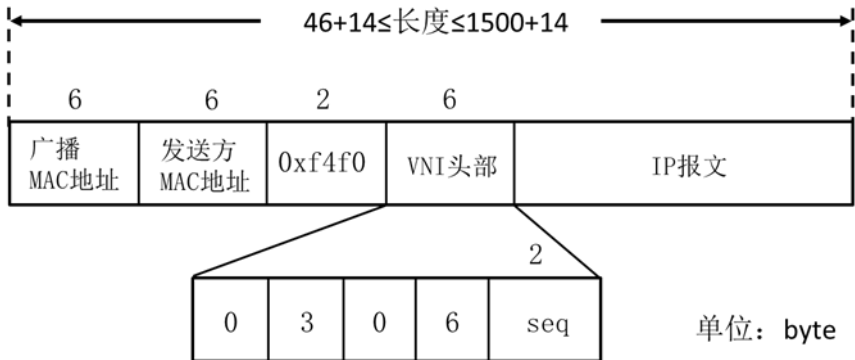


图 3-1 以太网帧部分结构（帧头和数据部分）

⑤发送数据帧。使用 `sendto()` 发送函数，借助构建好的 `socket` 将数据帧发送出去。同样需要借助 `sockaddr_ll` 地址类设置接收方。

以上即为发送任意自定义以太网帧的方案步骤。另外注意到本实验需要“截获”上层 IP 数据包来填充我们的自定义以太网帧。获取网络层向下发送的数据包同样可以用 `socket` 实现，只需要将 `socket` 创建为混杂模式，编写过滤函数、设置相应过滤条件筛选本机网络层下发的数据包（更一般的，还可以根据需求灵活改变过滤条件），其他配置与上述 `raw socket` 的使用

方案相似，使用与 `sendto()` 相对应的 `recvfrom()` 接收函数即可获取需要的数据包了。结合上述发送方案，即可实现 VNI 的发送部分功能。

发送统计、打印等比较简单，在主要发送功能实现完毕后，添加在模块合适的位置即可，在第四节程序实现中简单示出，不再赘述。

### 3.2 基于LKM的接收部分

接收报文就是报文在网络协议栈中流动并不断去头的过程。由上一节 Linux 网络协议栈原理部分可知，网卡驱动将报文通过 `netif_receive_skb()` 接口以 `sk_buff` 的形式送至内核网络层。在非桥接模式下，调用协议处理函数处理报文，如图 3-2 绿色方框中所示。因此，我们将自定义的 VNI 接口接收、处理报文的功能，作为一个 LKM，添加到内核协议模块集中。当报文被送到此处时，调用符合协议类型的协议处理函数，即我们编写、添加的 VNI 模块的接收处理函数，即可实现所需功能。

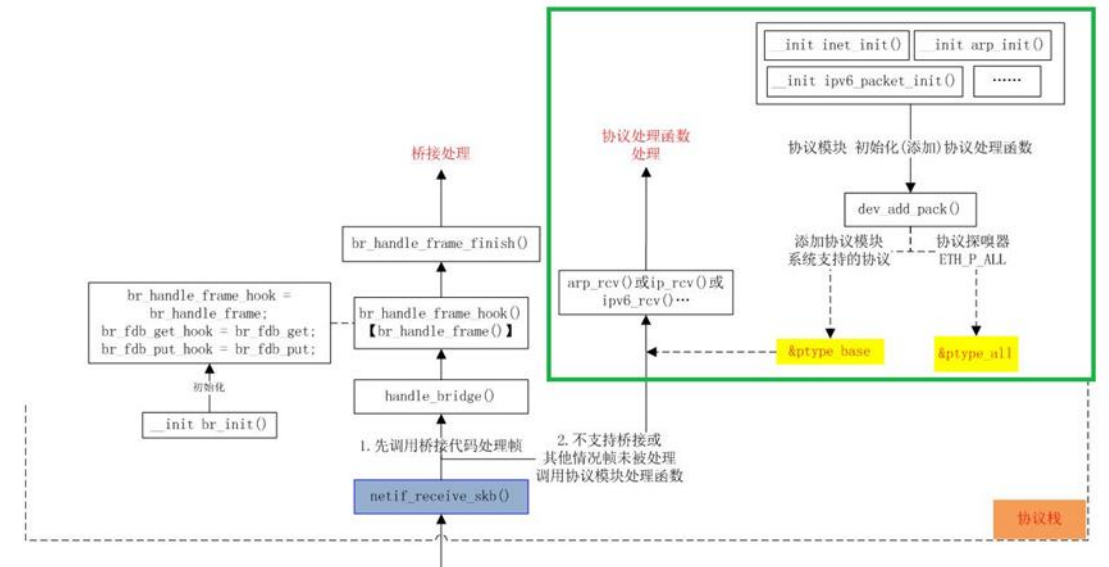


图 3-2 Linux 内核网络层对接收报文的处理

内核网络层的协议处理函数存储在链表 `&ptype_base` 上，报文到达后，内核在链表上查找匹配的协议、调用相应的协议处理函数。相对的，VNI 协议接收的处理也作为一个协议模块加入到 `&ptype_base` 链表中即可。具体步骤如下：

①创建 `packet_type` 数据结构。`packet_type` 结构供属于不同协议的数据包使用，设有协议类型、网络设备等属性作为筛选条件，以及一个对应的协议处理函数，在符合条件时被调用。具体的，我们将 2 字节 `type` 字段设置为 VNI 协议编号 `0xf4f0`，并为 `func` 属性初始化一个接收处理函数 `vni_rcv()`。

②编写接收处理函数 `vni_rcv()`，即 VNI 接收的功能函数。数据帧以 `sk_buff` 的结构送入，置于其 `data` 属性中。函数接收到数据帧后，判断目的 `ip` 是否为本机，若不是，则释放此 `sk_buff`，并返回丢弃结果。若是，则按照本实验的帧格式读出 VNI 头，再用 `skb_pull()` 方法以移动指针的方式“去除”VNI 头。

重新设置 `sk_buff` 的一些字段，如协议字段 `protocol` 设置为 `ETH_P_IP`（对应 IP 报文），报文类型 `pkt_type` 设置为 `PACKET_HOST`（对应发往本机）。最后同其他内置协议处理函数一样，用 `netif_rx()` 接口将处理完毕的 `sk_buff` 送往上层。

③整合，创建模块的加载、卸载接口。使用 `module_init()` 方法创建加载接口，在加载接口程序中将完成的 `packet_type` 数据结构添加到 `&ptype_base` 链表中；使用 `module_exit()` 方法创建卸载接口，在卸载接口程序中将 `packet_type` 数据移除 `&ptype_base` 链表。

以上即为 VNI 接收部分主要功能的实现过程。实现后，作为一个 LKM，可以使用 Linux 内核 `mod` 相关命令加载、使用和卸载。

另外，关于定时打印的功能，可以借助内核 `timer_list` 数据结构及相关的内置方法实现，同样在加载、卸载接口函数中分别添加、删除定时器即可。其他细节中，比较重要的会在第四节软件实现中给出。

## 4 软件实现

本节结合附件源码，简单介绍 VNI 的软件实现。软件实现实际上是上一节设计方案的编程过程，主脉络与方案中的“具体步骤”一致，依此路线进行介绍，同时补充一些重要的细节。分为发送部分和接收部分。

### 4.1 发送部分

从 Makefile 开始介绍本实现的逻辑，文件包含关系为 vnisend: vnisend.c macSendf.o macRecv.o，即 vnisend.c 为主函数文件，<macRecv.c macRecv.h>和<macSendf.c macSendf.h>为功能函数文件，分别提供“接收网络层下发的分组”和“向网卡发送自定义的 MAC 帧”所需的功能函数。

第一部分：常量、变量的初始化。如协议号 Ethertype 为 0xf4f0，网卡设备名 ens33，发送序号等。用 char 数组来建立“接收网络层下发的分组”、“向网卡发送自定义的 MAC 帧”所需的存储空间，因为一个 char 类型占 8bit，刚好与 1 字节数据相等。

第二部分：调用编写在功能函数文件中的 raw socket 创建、初始化函数 init\_macsocket\_send()。其功能为实现第三节所属步骤中的②和③，其中注意 socket 创建的 \_\_domain 取 AF\_PACKET，\_\_type 取 SOCK\_RAW，\_\_protocol 取 ETH\_P\_IP（本实验关注 IP 数据包）。函数返回 ifreq 结构体供发送时使用。

第三部分：调用编写在功能函数文件中的接收函数 mac\_rcv()循环侦听，当侦听到符合条件的，上层下传的 IP 报文时，启动下一部分。此处的“条件”可以自由设定，如空条件则为“本机 IP 模块下传的所有报文”。函数以 char 数组形式返回侦听到的报文。

为了演示方便，我们取一个比较强的条件，设定源 IP 为本机、目的 IP 为将要 ping 的对方 IP，仅关注对 ping 报文的动作。

第四部分：加 VNI 头、修改后调用编写在功能函数文件中的发送函数 mac\_send()发送。以 char 数组的形式填入 VNI 头部（4 字节学号和 2 字节序号）、IP 报文，序号自增，而后将此 char 数组送入 mac\_send()，同时的输入还有本机 MAC 地址作为源 MAC、广播地址(0xffffffff)作为目的 MAC、VNI 协议编号 0xf4f0（注意处理大端序/小端序），mac\_send()内部实现根据这些信息形成目的地址类、封装 MAC 帧，并调用 sendto()发送至以太网卡。

以上为发送部分的软件实现。操作上，因为发送部分是在用户空间使用的，运行比较方便，只需要在 shell 中执行 make 命令，自动根据 Makefile 生成.elf 可执行文件，再用./vnisend 命令运行即可。然后可以另开一个 shell，ping 另一个 IP。

### 4.2 接收部分

接收部分仅一个 vnirecv.c 源文件，且属于内核模块编写，结构比较八股。

第一部分：常量、变量的初始化。初始化一个 net\_device 类型的设备 vniRecv 以便于统计 VNI 接收到的报文数目，一个 timer\_list 类型的定时器 timer\_count 供“定时打印”功能的实现，以及其他辅助变量。

第二部分：统计接收数据包功能。统计数据包非常简单，只需要 vniRecv 固有的 stats.rx\_packets 属性（也可以另设一个全局变量）自增即可，此处为了代码的可维护性，还是将这一简单操作封装在了 vni\_rcv\_count()函数里，此函数将在第四部分 packet\_type 函数 vni\_rcv()接收到数据包时被调用。

第三部分：定时打印功能。借助 timer\_count 的 expires 属性和晶振相关的 jiffies、HZ 系统量即可实现定时触发，具体公式 timer\_count.expires = jiffies + period \* HZ，period 即为距下一次设定的时间，实验要求 1 分钟即为 60s，而后用 mod\_timer()设置定时触发。当前总接收报文数量减去上一次总接收报文数量即为当前周期接受数量，除以时间即可得到接收速率，但内核中浮点运算有些问题暂时没有搞清楚，故用两个整数分别表示速率浮点数的整数部分和小数部分。

第四部分：编写协议处理函数（packet\_type 函数）vni\_rcv()。注意到这个函数作为协议处理函数，需要遵守内核规定的输入和输出类型。函数获取到的 sk\_buff 类型的 skb 包含去除了 MAC 头的报文，skb->data 的前 6 字节即为我们的 VNI 头，可以获取、打印序号信息。“删除 VNI 头”的操作仿照内核其他协议处理函数用 skb\_pull\_rcsum()将 skb 数据头指针后移 6 字节即可。然后将 skb->protocol 属性改为 ETH\_P\_IP（IP 报文），skb->pkt\_type 属性改为



PACKET\_HOST（本机接受）。最后使用 `netif_rx()` 方法向上层递交 `skb`，调用前述统计接收的函数 `vni_recv_count()`，并返回 `NET_RX_SUCCESS`（接收成功）即可。

另外，由于协议处理函数的特殊性，自定义的 `vni_recv()` 也需要同内核其他协议处理函数一样，先读报文的 IP 头部分（从 `skb->data` 的第 7 个字节开始）判断本机是否为目的地，若是，执行前述 VNI 接收功能；若不是，则应丢弃该报文，调用 `kfree_skb()` 释放 `skb`，并返回 `NET_RX_DROP`（丢弃报文）。

第五部分：封装模块。这部分的格式是 Linux 内核规定好的，照模板填充即可。在 `__init` 函数中调用 `alloc_netdev()`、`register_netdev()` 添加设备，调用 `timer_setup()`、`add_timer()` 设置、添加定时器，调用 `dev_add_pack()` 添加我们的协议处理函数 `vni_recv()`；在 `__exit` 函数中调用 `dev_remove_pack()` 移除我们的协议处理函数，调用 `del_timer()` 删除定时器，调用 `unregister_netdev()`、`free_netdev()` 注销设备、释放空间。最后添加 `module_init()`、`module_exit()` 即可。

接收部分的使用是内核 LKM 的加载、卸载。使用 `make` 命令生成 `vnirecv.ko`，即可使用 `insmod`、`rmmod` 命令加载、卸载此模块。加载之后，模块会在内核识别、处理接收到的 VNI 协议（0xf4f0）的以太网帧。可以使用 `sudo cat /proc/kmsg` 命令查看内核打印，即可看到每分钟的定时打印信息。

## 5 测试与结果

本节对软件实现的 VNI 功能进行分步和整体测试，以验证效果。

测试环境：Linux 虚拟机版本为 Ubuntu-16.04.1；gcc 编译器版本 5.4.0-20160609。

辅助软件：Wireshark。



```
root@xyw-virtual-machine: /mnt/hgfs/share/vin/VIN_send
xyw@xyw-virtual-machine: /mnt/hgfs/share/vin/VIN_send$ sudo su
[sudo] xyw 的密码:
root@xyw-virtual-machine: /mnt/hgfs/share/vin/VIN_send# uname -a
Linux xyw-virtual-machine 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13
09:27:15 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
root@xyw-virtual-machine: /mnt/hgfs/share/vin/VIN_send# gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0-20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

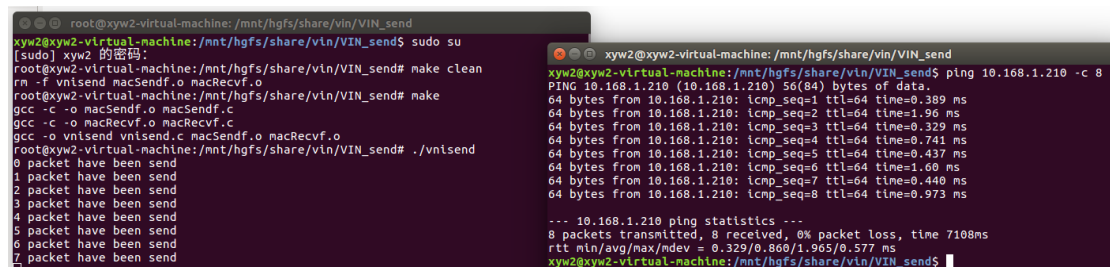
图 5-1 测试环境

用于实验的两台虚拟机分别称为 A、B。在演示中，虚拟机 A 作为发送端，IP 地址 10.168.1.185，MAC 地址 0x00:0x0c:0x29:0xb0:0x64:0xd3；虚拟机 B 作为接收端，IP 地址 10.168.1.210，MAC 地址 0x00:0x0c:0x29:0x08:0xab:0x10。

### 5.1 测试一：VNI 发送修改后的 ping 报文

本测试先验证模块修改、发送的报文符合题目要求，为显示界面看起来清爽，将软件实现中的条件设置为仅处理 A 发往 B 的 ping 报文。

在虚拟机 A 上，如第四节软件实现所述编译成功后，`sudo` 模式下运行 `vnisend.elf`，并 ping 虚拟机 B 的 IP 地址 8 次，如下图 5-2 所示，序号为 0 到 7 的 8 个报文被 VNI 成功发送。



```
root@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send
xyw2@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send$ sudo su
[sudo] xyw2 的密码:
root@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send# make clean
rm -f vnisend macSendf.o macRecv.o
root@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send# make
gcc -c -o macSendf.o macSendf.c
gcc -c -o macRecv.o macRecv.c
gcc -o vnisend vnisend.o macSendf.o macRecv.o
root@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send# ./vnisend
0 packet have been send
1 packet have been send
2 packet have been send
3 packet have been send
4 packet have been send
5 packet have been send
6 packet have been send
7 packet have been send

xyw2@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send$ ping 10.168.1.210 -c 8
PING 10.168.1.210 (10.168.1.210) 56(84) bytes of data:
64 bytes from 10.168.1.210: icmp_seq=1 ttl=64 time=0.389 ms
64 bytes from 10.168.1.210: icmp_seq=2 ttl=64 time=1.96 ms
64 bytes from 10.168.1.210: icmp_seq=3 ttl=64 time=0.329 ms
64 bytes from 10.168.1.210: icmp_seq=4 ttl=64 time=0.741 ms
64 bytes from 10.168.1.210: icmp_seq=5 ttl=64 time=0.437 ms
64 bytes from 10.168.1.210: icmp_seq=6 ttl=64 time=1.60 ms
64 bytes from 10.168.1.210: icmp_seq=7 ttl=64 time=0.440 ms
64 bytes from 10.168.1.210: icmp_seq=8 ttl=64 time=0.973 ms

--- 10.168.1.210 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7108ms
rtt min/avg/max/mdev = 0.329/0.860/1.965/0.577 ms
xyw2@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send$
```

图 5-2 ping 报文 8 次发送端结果

同时，我们用 Wireshark 抓包软件，设置条件抓取从本机送出的以太网帧，刚好 8 次，点开第一个和最后一个的帧数据，如图 5-3 和图 5-4 所示，可以看到目的 MAC 地址为广播地

址（红框内），源地址为虚拟机 A 的 MAC 地址（绿框内），VNI 协议编号为 0xf4f0（蓝框内），而后有 VNI 头部一次是 4 字节学号后四位 0x0306（黄线标注）和分别为 0 和 7 的分组序号（红线标注）。与要求完全一致，验证了发送报文的正确性。

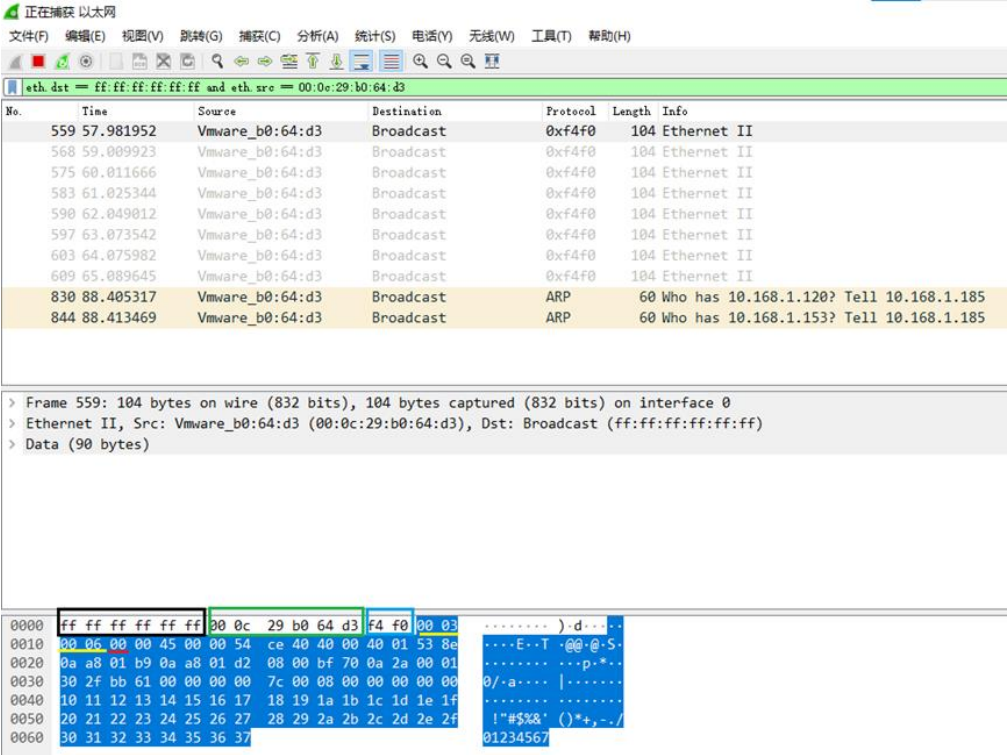


图 5-3 0 号帧

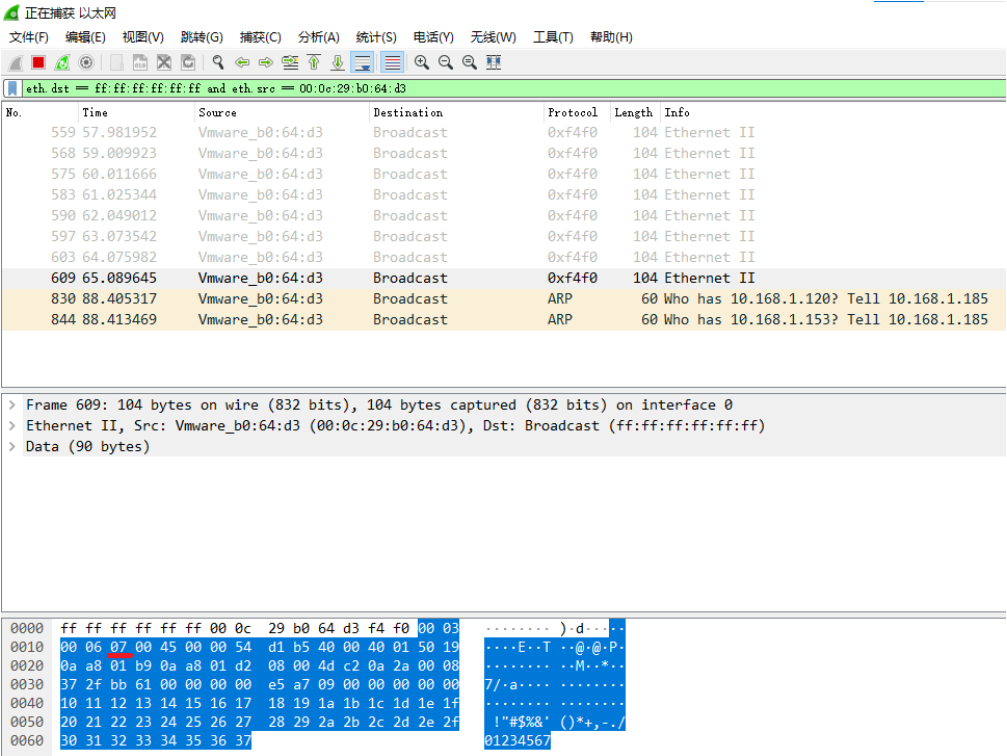


图 5-4 7 号帧

## 5.2 测试二：VNI接收0xf4f0协议报文

在作为接收端的虚拟机 B 上，如第四节软件实现所述编译成功后，使用 insmod 命令加



载 vnirecv 模块。之后可以用 lsmod 查看到模块已加载，如图 5-5 所示，同时可以用 ifconfig 命令看到新添加的虚拟设备，如图 5-6 所示。

```
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv# insmod vnirecv.ko
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv# lsmod
Module                               Size  Used by
vnirecv                             16384  0
vmw_vsock_vmci_transport             32768  2
vsock                               36864  3 vmw_vsock_vmci_transport
snd_ens1371                          28672  2
snd_ac97_codec                      131072 1 snd_ens1371
gameport                             16384 1 snd_ens1371
ac97_bus                             16384 1 snd_ac97_codec
snd_pcm                              98304 2 snd_ac97_codec,snd_ens1371
snd_seq_midi                         16384  0
```

图 5-5 vnirecv 模块加载

```
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv# ifconfig
ens33  Link encap:以太网 硬件地址 00:0c:29:08:ab:10
       inet 地址:10.168.1.210 广播:10.168.1.255 掩码:255.255.255.0
       inet6 地址: fe80::6140:9607:d6a:162d/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST MTU:1500 跃点数:1
       接收数据包:25529 错误:0 丢弃:1 过载:0 帧数:0
       发送数据包:1383 错误:0 丢弃:0 过载:0 载波:0
       碰撞:0 发送队列长度:1000
       接收字节:4749396 (4.7 MB) 发送字节:244153 (244.1 KB)

lo     Link encap:本地环回
       inet 地址:127.0.0.1 掩码:255.0.0.0
       inet6 地址: ::1/128 Scope:Host
       UP LOOPBACK RUNNING MTU:65536 跃点数:1
       接收数据包:850 错误:0 丢弃:0 过载:0 帧数:0
       发送数据包:850 错误:0 丢弃:0 过载:0 载波:0
       碰撞:0 发送队列长度:1000
       接收字节:76206 (76.2 KB) 发送字节:76206 (76.2 KB)

vniRecv0 Link encap:以太网 硬件地址 00:00:00:00:00:00
        UP BROADCAST RUNNING MULTICAST MTU:1500 跃点数:1
        接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
        发送数据包:0 错误:0 丢弃:0 过载:0 载波:0
        碰撞:0 发送队列长度:1000
```

图 5-6 查看设备 vniRecv0

仍在虚拟机 A 上运行 VNI 发送程序，不断 ping 报文，在虚拟机 B 上使用 `sudo cat /proc/kmsg` 命令或 `dmesg` 命令，可以看到累计收到的 VNI 报文及速率（pps），如图 5-7 所示。

```
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv
<4>[ 5064.524541] vni header: 00:03:00:06:25
<4>[ 5065.527651] vni header: 00:03:00:06:26
<4>[ 5066.530167] vni header: 00:03:00:06:27
<4>[ 5069.536864] vni header: 00:03:00:06:30
<4>[ 5071.541596] vni header: 00:03:00:06:32
<4>[ 5073.545235] vni header: 00:03:00:06:34
<4>[ 5074.574605] vni header: 00:03:00:06:35
<4>[ 5075.576637] vni header: 00:03:00:06:36
<4>[ 5077.008583] total received vni packets : 30
<4>[ 5077.008681] receiving rate of vni packet : 1.0
```

图 5-7 接收模块打印信息

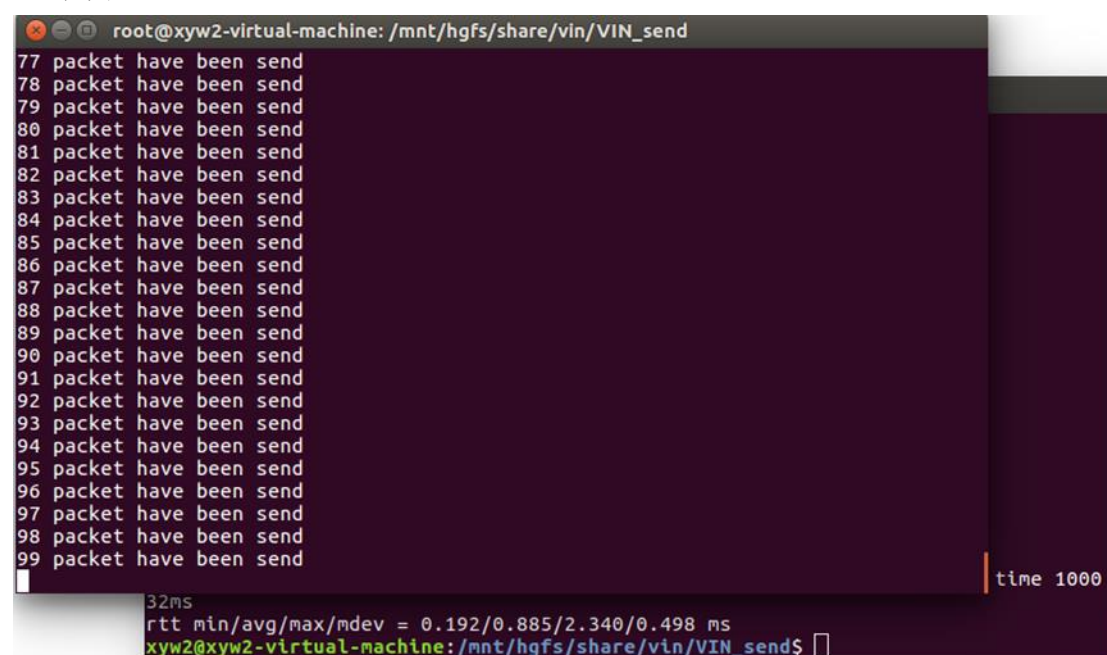
可以看到，VNI 接收模块将接收到的 VNI 头部信息打印了出来，学号后四位 0x0306 不



变，序号递增（但打印有些随机丢失，还不清楚原因，可能与内核线程、中断有关），如图 5-7 绿框部分。同时每隔一段时间打印累计接收报文数量和速率，如图 5-7 蓝框部分，累计接收 30 个 VNI 报文，速率 1.0pps，因为发送端现在仅筛选 ping 报文做 VNI 处理，而 ping 的速率约为 1 秒 1 次。综上，接收部分验证无误。

### 5.3 测试三：仅关注 ping 报文条件下，100 次 ping 收发实验

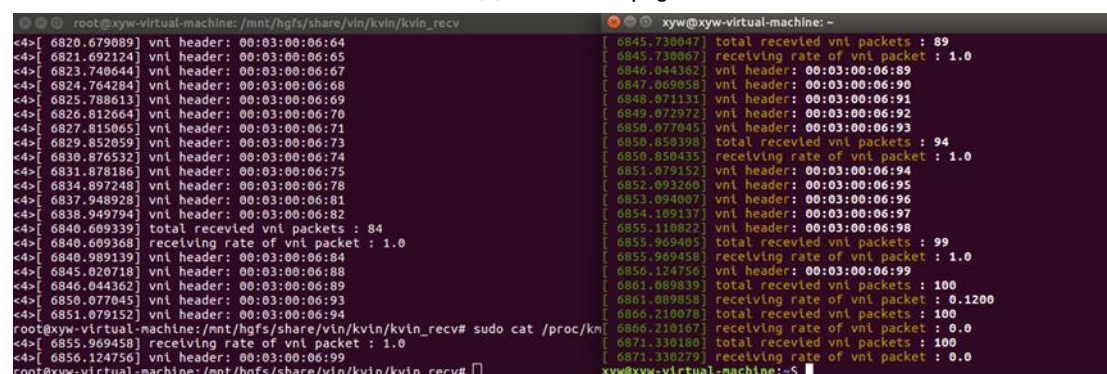
由于仅针对 ping 报文的实验效果比较清晰，在最后的实验前，进行仅针对 ping 报文的 100 次收发实验，即发送端发送程序设定条件为“仅在 IP 包为 ping 报文时，依照题目 VNI 的要求进行修改”。同前两个测试，虚拟机 A 和 B 加载上接收模块、运行发送程序，结果如图 5-8 和图 5-9。



```
root@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send
77 packet have been send
78 packet have been send
79 packet have been send
80 packet have been send
81 packet have been send
82 packet have been send
83 packet have been send
84 packet have been send
85 packet have been send
86 packet have been send
87 packet have been send
88 packet have been send
89 packet have been send
90 packet have been send
91 packet have been send
92 packet have been send
93 packet have been send
94 packet have been send
95 packet have been send
96 packet have been send
97 packet have been send
98 packet have been send
99 packet have been send

time 1000
32ms
rtt min/avg/max/mdev = 0.192/0.885/2.340/0.498 ms
xyw2@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send$
```

图 5-8 虚拟机 A（发送端）ping100 结果



```
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv
<4>[ 6820.679089] vni header: 00:03:00:06:64
<4>[ 6821.692124] vni header: 00:03:00:06:65
<4>[ 6823.740644] vni header: 00:03:00:06:67
<4>[ 6824.764284] vni header: 00:03:00:06:68
<4>[ 6825.788613] vni header: 00:03:00:06:69
<4>[ 6826.812664] vni header: 00:03:00:06:70
<4>[ 6827.815865] vni header: 00:03:00:06:71
<4>[ 6829.852899] vni header: 00:03:00:06:73
<4>[ 6830.876532] vni header: 00:03:00:06:74
<4>[ 6831.878186] vni header: 00:03:00:06:75
<4>[ 6834.897248] vni header: 00:03:00:06:78
<4>[ 6837.948928] vni header: 00:03:00:06:81
<4>[ 6838.949794] vni header: 00:03:00:06:82
<4>[ 6840.609339] total received vni packets : 84
<4>[ 6840.609368] receiving rate of vni packet : 1.0
<4>[ 6840.989139] vni header: 00:03:00:06:84
<4>[ 6845.020718] vni header: 00:03:00:06:88
<4>[ 6846.044362] vni header: 00:03:00:06:89
<4>[ 6850.077045] vni header: 00:03:00:06:93
<4>[ 6851.079152] vni header: 00:03:00:06:94
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv# sudo cat /proc/kn
<4>[ 6855.969458] receiving rate of vni packet : 1.0
<4>[ 6856.124756] vni header: 00:03:00:06:99
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv#

xyw@xyw-virtual-machine: -
[ 6845.730047] total received vni packets : 89
[ 6845.730067] receiving rate of vni packet : 1.0
[ 6846.044362] vni header: 00:03:00:06:89
[ 6847.069058] vni header: 00:03:00:06:90
[ 6848.071131] vni header: 00:03:00:06:91
[ 6849.072972] vni header: 00:03:00:06:92
[ 6850.077045] vni header: 00:03:00:06:93
[ 6850.850398] total received vni packets : 94
[ 6850.850435] receiving rate of vni packet : 1.0
[ 6851.079152] vni header: 00:03:00:06:94
[ 6852.093260] vni header: 00:03:00:06:95
[ 6853.094007] vni header: 00:03:00:06:96
[ 6854.109137] vni header: 00:03:00:06:97
[ 6855.110822] vni header: 00:03:00:06:98
[ 6855.969405] total received vni packets : 99
[ 6855.969458] receiving rate of vni packet : 1.0
[ 6856.124756] vni header: 00:03:00:06:99
[ 6861.089039] total received vni packets : 100
[ 6861.089050] receiving rate of vni packet : 0.1200
[ 6866.210070] total received vni packets : 100
[ 6866.210167] receiving rate of vni packet : 0.0
[ 6871.330180] total received vni packets : 100
[ 6871.330279] receiving rate of vni packet : 0.0
xyw@xyw-virtual-machine:~$
```

图 5-9 虚拟机 B（接收端）ping100 结果

其中接收端为了容易看到打印效果，设置的打印“累计收包及速率”的周期是 5s 而非 1 分钟。与前两个测试的分析一致，不再赘述，效果符合预期。

### 5.4 测试四：空条件下，100 次 ping 收发实验（即题目要求的实验）

经过前面的铺垫测试，最后，我们设置 VNI 发送程序中条件为空，即任何从虚拟机 A 内核 IP 模块发出的报文都进行 VNI 的处理，并且 ping 虚拟机 B 共 100 次，且统计打印的周期改为题设 1 分钟一次，结果如图 5-10 和图 5-11。

从图 5-10 可以看出，ping 报文 100 次，而 VNI 在此期间却处理了共 299 个报文，这是因为除 ping 报文外，虚拟机 A 还会有其他业务在向外发送 IP 报文，因此 ping 的 100 个报文如同散落在发送流中，其序号也不再是连续的从 0 到 99，这一点可以从图 5-11 接收端提取出的、间断的 VNI 头部序号字段得到验证。

```
root@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send
277 packet have been send
278 packet have been send
279 packet have been send
280 packet have been send
281 packet have been send
282 packet have been send
283 packet have been send
284 packet have been send
285 packet have been send
286 packet have been send
287 packet have been send
288 packet have been send
289 packet have been send
290 packet have been send
291 packet have been send
292 packet have been send
293 packet have been send
294 packet have been send
295 packet have been send
296 packet have been send
297 packet have been send
298 packet have been send
299 packet have been send
time 1001
rtt min/avg/max/mdev = 0.223/0.789/2.945/0.503 ms
xyw2@xyw2-virtual-machine: /mnt/hgfs/share/vin/VIN_send$
```

图 5-10 100 次 ping 实验发送端打印

```
root@xyw-virtual-machine: /mnt/hgfs/share/vin/kvin/kvin_recv
<4>[ 8372.986566] vni header: 00:03:00:06:267
<4>[ 8372.986931] vni header: 00:03:00:06:268
<4>[ 8372.986966] vni header: 00:03:00:06:269
<4>[ 8373.999360] vni header: 00:03:00:06:270
<4>[ 8375.026423] vni header: 00:03:00:06:273
<4>[ 8375.027402] vni header: 00:03:00:06:274
<4>[ 8375.027411] vni header: 00:03:00:06:275
<4>[ 8376.026745] vni header: 00:03:00:06:276
<4>[ 8376.027779] vni header: 00:03:00:06:278
<4>[ 8377.027763] vni header: 00:03:00:06:279
<4>[ 8377.028233] vni header: 00:03:00:06:280
<4>[ 8377.028249] vni header: 00:03:00:06:281
<4>[ 8378.031820] vni header: 00:03:00:06:284
<4>[ 8379.056032] vni header: 00:03:00:06:285
<4>[ 8379.058417] vni header: 00:03:00:06:286
<4>[ 8379.058422] vni header: 00:03:00:06:287
<4>[ 8380.057987] vni header: 00:03:00:06:288
<4>[ 8380.058757] vni header: 00:03:00:06:290
<4>[ 8381.059980] vni header: 00:03:00:06:291
<4>[ 8382.063175] vni header: 00:03:00:06:294
<4>[ 8383.087711] vni header: 00:03:00:06:297
xyw@xyw-virtual-machine: ~
3.422624] vni header: 00:03:00:06:243
3.423189] vni header: 00:03:00:06:244
3.423243] vni header: 00:03:00:06:245
3.437340] vni header: 00:03:00:06:246
3.437580] vni header: 00:03:00:06:247
3.437792] vni header: 00:03:00:06:248
3.629243] total received vni packets : 83
3.629344] receiving rate of vni packet : 1.0
3.461204] vni header: 00:03:00:06:249
3.462129] vni header: 00:03:00:06:250
3.462218] vni header: 00:03:00:06:251
1.463628] vni header: 00:03:00:06:252
1.464509] vni header: 00:03:00:06:253
1.464881] vni header: 00:03:00:06:254
2.466351] vni header: 00:03:00:06:255
2.467199] vni header: 00:03:00:06:256
2.467786] vni header: 00:03:00:06:257
3.468675] vni header: 00:03:00:06:258
3.469403] vni header: 00:03:00:06:259
```

图 5-11 100 次 ping 实验接收端打印

进一步，设定 Wireshark 抓包条件为源 MAC 地址为虚拟机 A 的 MAC 地址，协议类型为 VNI 协议 0xf4f0，如图 5-12 所示。在此期间抓到的最后一个包的序号为 0x012b（图 5-12 蓝框内所示，因为 x86 架构为小端序，short 型的序号两字节被反序，实际值应为“01”在前“2b”在后），十进制即为 299，与收发结果相符。

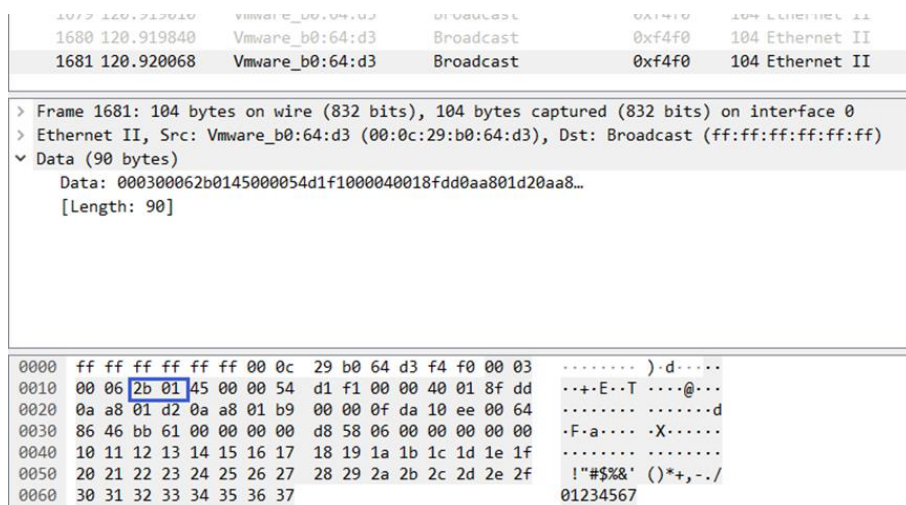


图 5-12 VNI 发送的最后一帧

综上，我们完成了 VNI 模块的收发验证。

## 6 结束语

本文介绍了一种 VNI 的编程实现。发送部分基于 raw socket 实现，可以自定义条件规定模块处理网络层下发的何种报文，还可以将内容任意的以太帧发送到网口；接收部分以注册内核 LKM 的形式实现。并且进行了软件运行测试、抓包对比和结果分析。

### 参考文献：

- [1] LEFFLER S J, JOY W N, FABRY R S. 4.2BSD Networking Implementation Notes[J]. : 31.
- [2] A Guide to Using Raw Sockets - open source for you[EB/OL]. [2021-12-16].  
<https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/>.
- [3] snippet/raw-socket-demystified.txt at master · xgfone/snippet[EB/OL]. GitHub, [2021-12-16].  
<https://github.com/xgfone/snippet>.
- [4] Linux可加载内核模块（LKM）\_liujianqing的专栏-CSDN博客\_lkm是什么意思[EB/OL]. [2021-12-16].  
<https://blog.csdn.net/zhaqiwen/article/details/8288472>.