

计算机网络课程实验报告

实验 2 - 数据帧与 IP 包分析

实验时间: 20th Oct, 2016

1) 实验内容与目的

本次实验将在 Windows 环境与 Linux 环境下使用相关网络工具进行数据帧与 IP 包的分析，具体由以下几部分实验组成：

- (1) 通过在 Windows 环境下使用 Wireshark 捕获并分析一系列数据帧与报文，掌握以太网 MAC 帧与 IP 数据报的构成，了解其各字段含义，初步了解网络流量捕获与分析的有关方法，并掌握 ARP/ICMP 协议的请求/响应机理；
- (2) 在 CentOS(VM) 环境下使用 sendip 工具构造、封装简单的 IP 数据报，并在 Windows 下使用 Wireshark 进行捕获分析；
- (3) 编程模拟 IP 数据报首部的封装。

2) 实验过程与分析

实验 1. 使用 Wireshark 捕获并分析数据

1.1) 记录和解释一个 MAC 帧

在启动 Wireshark 时需要指定进行监听的网络接口，实验室硬件环境只有一个网络接口，因此我们在这个接口上进行捕获即可。

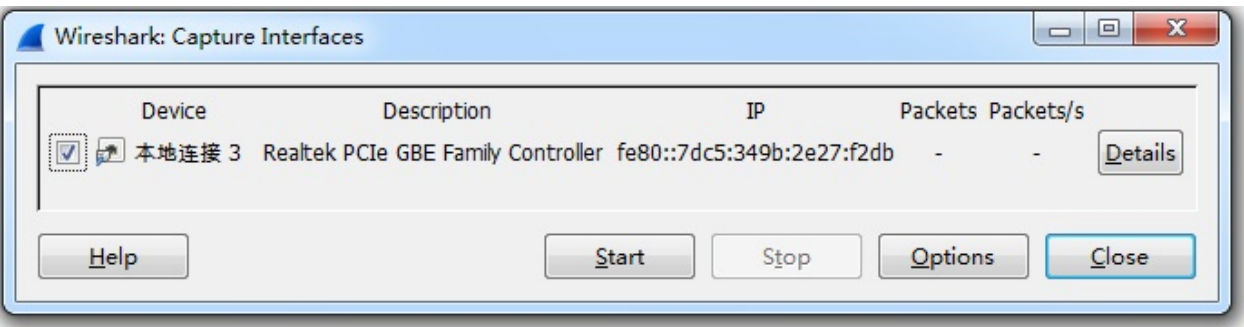


Fig 1.1 Setting up Wireshark capture interface

在 Wireshark 开始监听网络流量后，我们可以发起一个网络活动来捕获对应的 MAC 帧。
如图 1.2 所示，这里我们发起了一个 ping 请求。

```
C:\Users\Netlab>ping 202.89.233.103

正在 Ping 202.89.233.103 具有 32 字节的数据:
来自 202.89.233.103 的回复: 字节=32 时间=48ms TTL=115
来自 202.89.233.103 的回复: 字节=32 时间=48ms TTL=115
来自 202.89.233.103 的回复: 字节=32 时间=50ms TTL=115
来自 202.89.233.103 的回复: 字节=32 时间=48ms TTL=115

202.89.233.103 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间<以毫秒为单位>:
        最短 = 48ms, 最长 = 50ms, 平均 = 48ms
```

Fig 1.2 Pinging IPv4 address 202.89.233.103

因为在一台主机上通常会有很多不同的网络活动在同时进行，因而 Wireshark 可能捕获到大量与我们发起的网络活动无关的流量包。我们可以使用过滤器来过滤出我们所关注的信息，在这里我们可以通过目标 IP 地址实现过滤： `ip.dst == 202.89.233.103`，可以得到下面的帧信息：

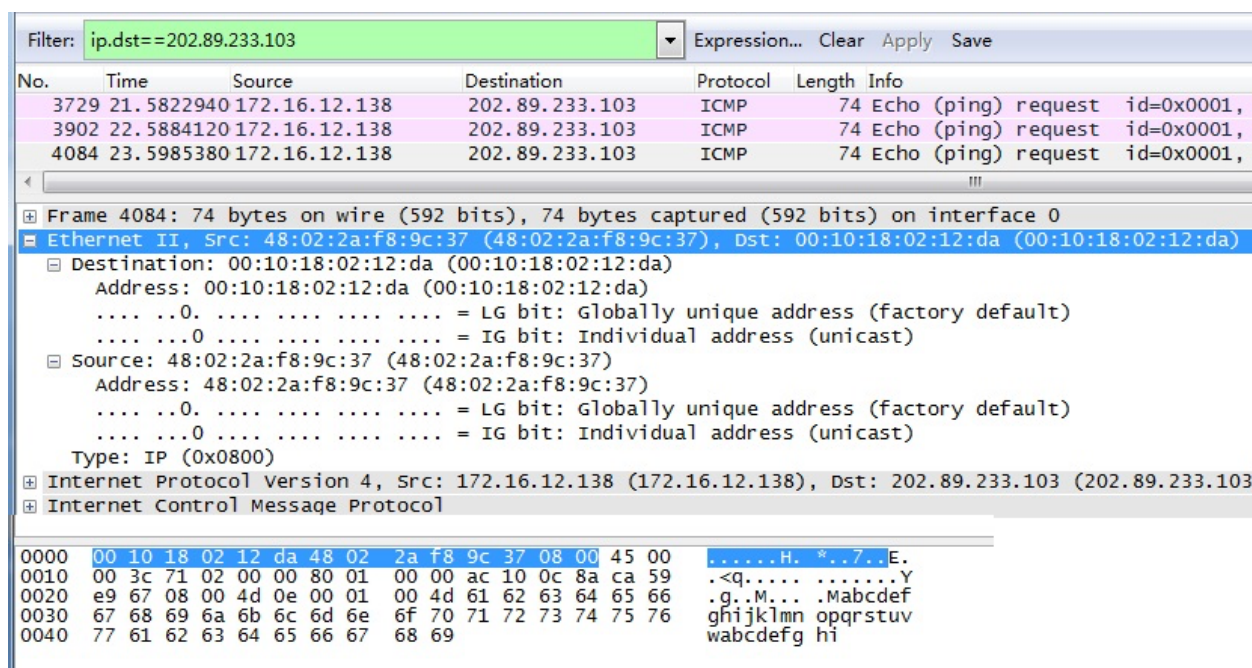


Fig 1.3 Corresponding frame of the Ping action (showing MAC frame header section)

数据链路层把网络层交付的 IP 数据报加上帧首部、帧尾部与边界符等构成 MAC 帧后发送到数据链路上，图 1.3 中蓝色高亮部分便是 MAC 帧的首部部分，之后才是 IP 数据报的正式内容。

可以观察到该 MAC 帧首部共有 14 个字节，其详细组成如下：

1. Destination: 00:10:18:02:12:da
代表接收数据帧的目标节点 MAC 地址，长度为 6 个字节 (48 位)。
2. Source: 48:02:2a:f8:9c:37
代表发送数据帧的源节点 MAC 地址，即本机 MAC 地址，长度为 6 个字节。
3. Type: IP (0x0800)
标识出以太网帧所携带的上层数据类型，0x0800 表示上层为 IP 协议。本部分长度为 2 个字节。
MAC 帧尾部为 4 字节长的 FCS 校验和，在 Wireshark 中未予以显示。

以太网适配器 本地连接 3:

```
连接特定的 DNS 后缀 . . . . . :  
描述 . . . . . : Realtek PCIe GBE Family Controller  
物理地址. . . . . : 48-02-2A-F8-9C-37  
DHCP 已启用 . . . . . : 否  
自动配置已启用 . . . . . : 是  
本地连接 IPv6 地址. . . . . : fe80::7dc5:349b:2e27:f2db%18<首选>  
IPv4 地址 . . . . . : 172.16.12.138<首选>  
子网掩码 . . . . . : 255.255.252.0  
默认网关 . . . . . : 172.16.12.1  
DHCPv6 IAID . . . . . : 407372330  
DHCPv6 客户端 DUID . . . . . : 00-01-00-01-18-50-62-D3-68-05-CA-0F-70-A1  
  
DNS 服务器 . . . . . : 210.34.0.14  
                        210.34.0.18  
TCP/IP 上的 NetBIOS . . . . . : 已启用
```

Fig 1.4 Local network configurations

通过 `ipconfig /all` 命令可以看到该接口的物理地址与 Wireshark 捕获到的 MAC 帧中的对应字段一致。

- 问题：为什么以太网中 MAC 帧数据字段最小长度为 46 字节？
 - 以太网 MAC 帧受“碰撞检测”限制，规定了 MAC 帧最小长度为 64 字节，而 MAC 帧首部含有 14 字节，尾部含有 4 字节，故中间的数据字段最小长度为 $64 - 14 - 4 = 46$ (bytes)。

1.2) 记录和解释一个 IP 数据报

采取同 1.1 中类似的方法，对 202.89.233.103 发起 Ping 请求，使用 Wireshark 过滤截获的 IP 数据报。

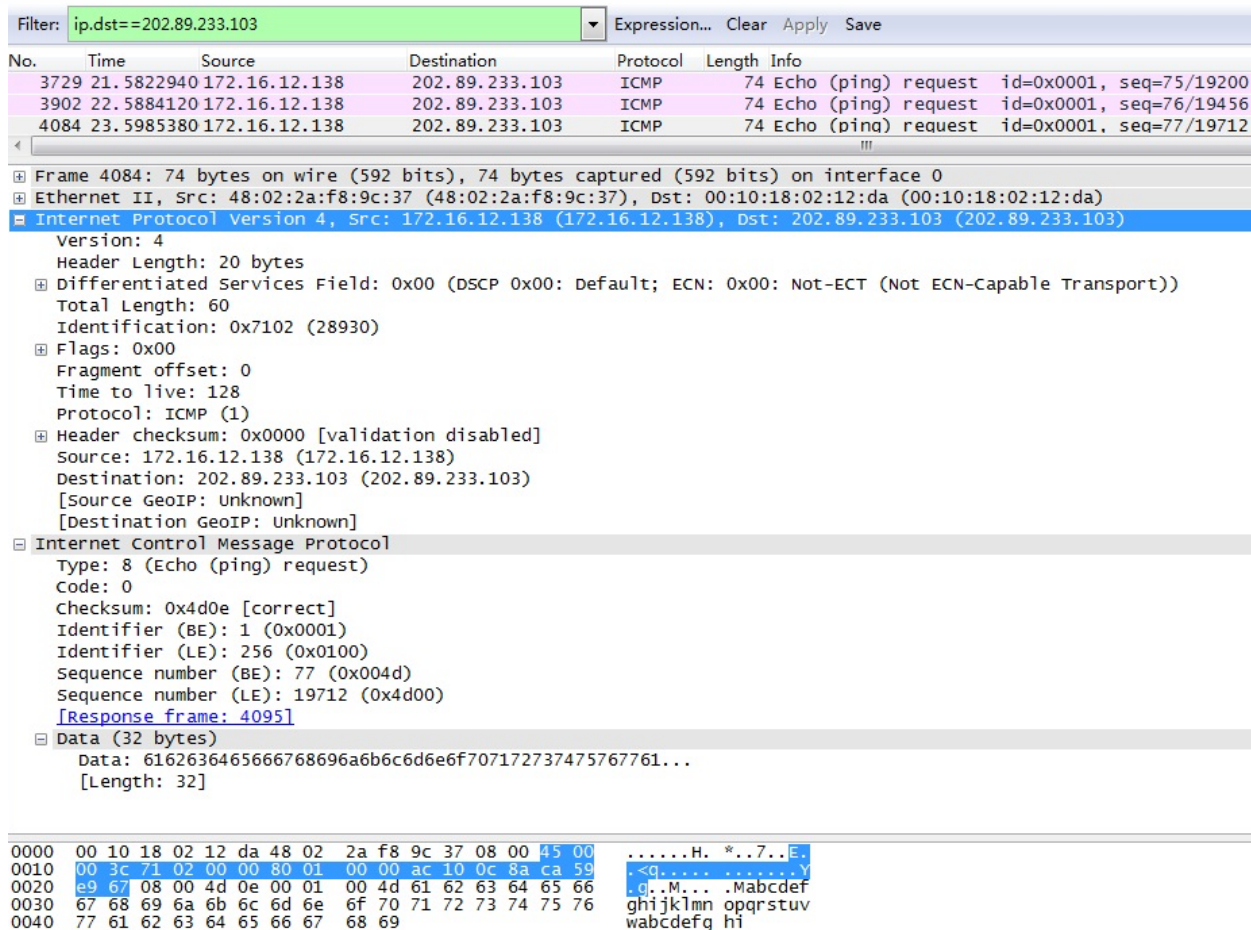


Fig 1.5 Header section of IP packet

图 1.5 中蓝色高亮部分即为截获的 IP 数据报首部，可以观察到 IP 数据报首部共有 20 个字节，其详细组成如下：

1. Version: 4

标注所使用的 IP 协议的版本，一般为 IPv4 或 IPv6，长度为 4 位。4 说明使用的 IP 版本为 IPv4。

2. Header Length: 20 bytes

标明 IP 数据报首部的长度，长度为 4 位。这里其数据值为 5，其单位为 32 位字，故表示数据报首部长为 $5 * 32 \text{ bit} = 20 \text{ bytes}$ 。

1. Differentiated Services Field: 0x00

区分服务字段，一般不使用。长度为 1 个字节 (8 位)。

2. Total Length: 60

标明 IP 数据报首部与数据之和的总长度，占 2 个字节。此处说明该数据报总长度为 60 字节。

3. Identification: 0x7102

标识，占 2 个字节。IP 软件维持一个计数器，每产生一个数据报计数器就加1，并将此值赋值给标识字段，但它不起“序号”的作用。相同的标识字段的值使分片后的各数据片最后能正确地重装成为原来的数据报。

4. Flags: 0x00

标志字段，占 3 位，但目前只有后两位有意义。最低位 $MF = 1$ 时代表后面“还有分片”的数据报，这里 $MF = 0$ 说明没有分片。中间一位 $DF = 1$ 时代表“不能分片”，此处 $DF = 0$ 代表可以分片。

5. Fragment offset: 0

片偏移，占 13 位。片偏移指出较长的分组在分片后其中某片在原分组中的相对位置，为 0 时代表该分组在数据的最前面。一般以 8 个字节作为偏移单位。

6. Time to live: 128

生存时间，占 1 字节，表明了数据报在网络中的寿命。128 代表该数据报最多可以经过路由器跳转 128 次。

1. Protocol: ICMP (1)

协议，占 1 字节。表明该数据报所使用的协议。此处 1 代表本数据报使用的是 ICMP 协议。

2. Header checksum: 0x0000 [validation disabled]

首部检验和，占 2 字节。负责检验数据报的首部数据的准确性，此处未启用。

3. Source

源地址，占 4 字节。发送该数据报的源节点的 IP 地址。

4. Destination

目的地址，占 4 字节。该数据报的目的 IP 地址。

1.3) 记录和解释一系列 ARP 报文

由于操作系统储存有 ARP 高速缓存以提高网络利用效率（使用 `arp -a` 命令可查询当前缓存），所以在准备捕获 ARP 报文之前需要先清空 ARP 高速缓存（`arp -d` 命令）方能使主机发起 ARP 请求。

Filter: arp		Expression... Clear Apply Save				
No.	Time	Source	Destination	Protocol	Length	Info
416	2.51136700	48:02:2a:f8:9c:98	Broadcast	ARP	60	who has 172.16.12.1? Tell 172.16.12.141
523	3.15913600	48:02:2a:f8:9c:37	Broadcast	ARP	42	who has 172.16.12.1? Tell 172.16.12.138
524	3.15993400	00:10:18:02:12:da	48:02:2a:f8:9c:37	ARP	60	172.16.12.1 is at 00:10:18:02:12:da
526	3.16071100	00:10:18:02:12:da	Broadcast	ARP	60	who has 172.16.12.138? Tell 172.16.12.1
528	3.16072300	48:02:2a:f8:9c:37	00:10:18:02:12:da	ARP	42	172.16.12.138 is at 48:02:2a:f8:9c:37
565	3.38931900	00:10:18:02:12:da	Broadcast	ARP	60	who has 172.16.12.128? Tell 172.16.12.1

Frame 524: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0	
Ethernet II, Src: 00:10:18:02:12:da (00:10:18:02:12:da), Dst: 48:02:2a:f8:9c:37 (48:02:2a:f8:9c:37)	
Destination: 48:02:2a:f8:9c:37 (48:02:2a:f8:9c:37)	Address: 48:02:2a:f8:9c:37 (48:02:2a:f8:9c:37)
....0. = LG bit: Globally unique address (factory default)	
....0. = IG bit: Individual address (unicast)	
Source: 00:10:18:02:12:da (00:10:18:02:12:da)	Address: 00:10:18:02:12:da (00:10:18:02:12:da)
....0. = LG bit: Globally unique address (factory default)	
....0. = IG bit: Individual address (unicast)	
Type: ARP (0x0806)	
Padding: 00000000000000000000000000000000	
Address Resolution Protocol (reply)	
Hardware type: Ethernet (1)	
Protocol type: IP (0x0800)	
Hardware size: 6	
Protocol size: 4	
Opcode: reply (2)	
Sender MAC address: 00:10:18:02:12:da (00:10:18:02:12:da)	
Sender IP address: 172.16.12.1 (172.16.12.1)	
Target MAC address: 48:02:2a:f8:9c:37 (48:02:2a:f8:9c:37)	
Target IP address: 172.16.12.138 (172.16.12.138)	

Fig 1.8 Corresponding ARP reply

可以看到捕获到的 ARP 报文长度为 28 字节，由以下部分组成：

1. Hardware type: Ethernet (1)

硬件类型，占 2 个字节，表示硬件地址的类型。此处其值为 1，表示硬件类型为以太网。

1. Protocol type: IP (0x0800)

协议类型，占 2 个字节，表示要映射的协议地址类型。此处其值为 0x0800，表示映射 IP 协议地址。

2. Hardware size: 6

硬件地址长度，占 1 个字节，指出硬件地址的长度，以字节为单位。对于以太网上 IP 地址的 ARP 请求或应答来说，硬件地址长度为 6 个字节，即 MAC 地址。

1. Protocol size: 4

协议地址长度，占 1 个字节，指出协议地址的长度，以字节为单位。对于以太网上 IP 地址的 ARP 请求或应答来说，协议地址长度为 4 个字节，即 IP 地址；

2. Opcode: reply (2)

操作类型，占 2 个字节，指出 ARP 操作的类别。1 表示 ARP request, 2 表示 ARP reply.

1. Sender MAC address

发送端 MAC 地址，占 6 个字节，指明发送方设备的硬件 MAC 地址。

2. Sender IP address

发送端 IP 地址，占 4 个字节，指明发送方的 IP 地址。

3. Target MAC address

目的 MAC 地址，占 6 个字节，指明接受方设备的硬件 MAC 地址。

4. Target IP address

目的 IP 地址，占 4 个字节，指明接受方的 IP 地址。

- 这一对 ARP 报文中，请求报文采用 **广播** 方式向同一局域网内的设备发起询问，同时报告自己的硬件地址；与请求报文中的目的地址一致的设备收到请求后记录下来源的硬件地址与 IP 地址信息，同时采用 **单播** 方式向请求来源发出回复报文报告自己的硬件地址与 IP 地址信息。
- **问题：Ping 局域网内部与局域网外部的计算机所产生的 ARP 报文有何不同？**
 - 主机向局域网内部的计算机发起网络请求时，主机所广播的请求报文中目的 IP 地址与所请求的 IP 地址一致，获取到目的主机地址后两者便可直接通信；而主机向局域网外部的计算机发起网络请求时，主机所广播的请求报文中目的 IP 地址均为局域网路由器地址，这时主机先通过询问路由器所在地址建立与路由器的连接，再通过与路由器通信来实现与外部计算机的通信。

1.4) 记录和解释一系列 ICMP 报文

采用类似于上面三个实验的方法对 114.114.114.114 发起 Ping 请求并用 Wireshark 过滤截获的 ICMP 报文。

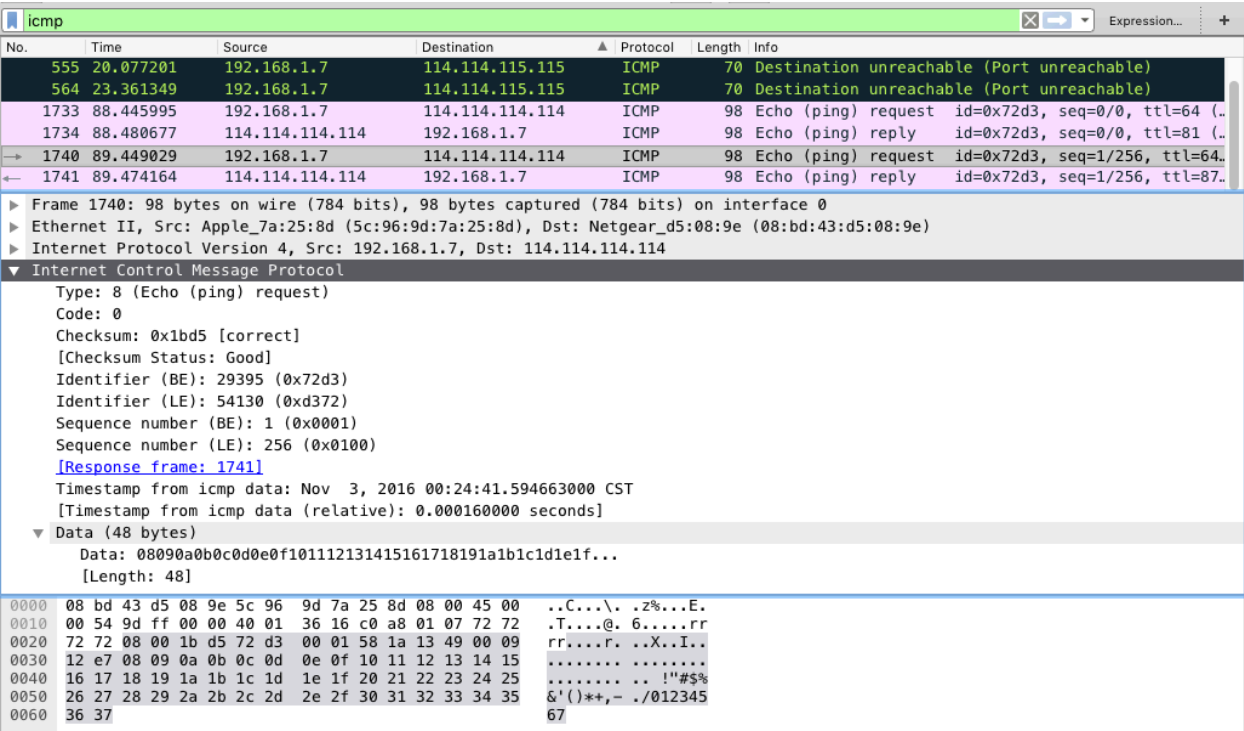


Fig 1.9 ICMP request

No.	Time	Source	Destination	Protocol	Length	Info
555	20.077201	192.168.1.7	114.114.115.115	ICMP	70	Destination unreachable (Port unreachable)
564	23.361349	192.168.1.7	114.114.115.115	ICMP	70	Destination unreachable (Port unreachable)
1733	88.445995	192.168.1.7	114.114.114.114	ICMP	98	Echo (ping) request id=0x72d3, seq=0/0, ttl=64 (..
1734	88.480677	114.114.114.114	192.168.1.7	ICMP	98	Echo (ping) reply id=0x72d3, seq=0/0, ttl=81 (..
→ 1740	89.449029	192.168.1.7	114.114.114.114	ICMP	98	Echo (ping) request id=0x72d3, seq=1/256, ttl=64..
← 1741	89.474164	114.114.114.114	192.168.1.7	ICMP	98	Echo (ping) reply id=0x72d3, seq=1/256, ttl=87..

▶ Frame 1741: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
 ▶ Ethernet II, Src: Netgear_d5:08:9e (08:bd:43:d5:08:9e), Dst: Apple_7a:25:8d (5c:96:9d:7a:25:8d)
 ▶ Internet Protocol Version 4, Src: 114.114.114.114, Dst: 192.168.1.7

▼ Internet Control Message Protocol

Type: 0 (Echo (ping) reply)
 Code: 0
 Checksum: 0x23d5 [correct]
 [Checksum Status: Good]
 Identifier (BE): 29395 (0x72d3)
 Identifier (LE): 54130 (0xd372)
 Sequence number (BE): 1 (0x0001)
 Sequence number (LE): 256 (0x0100)
[\[Request frame: 1740\]](#)
 [Response time: 25.135 ms]
 Timestamp from icmp data: Nov 3, 2016 00:24:41.594663000 CST
 [Timestamp from icmp data (relative): 0.025295000 seconds]

▼ Data (48 bytes)

Data: 08090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f...

```

0000 5c 96 9d 7a 25 8d 08 bd 43 d5 08 9e 08 00 45 00 \..z%... C....E.
0010 00 54 2f 0d 00 00 57 01 8e 08 72 72 72 72 c0 a8 .T/...W. ....rrrr..
0020 01 07 00 00 23 d5 72 d3 00 01 58 1a 13 49 00 09 ...#.r. ..X..I..
0030 12 e7 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 .....
0040 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 ..... !"#%$
0050 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 &'()*+,-./012345
0060 36 37 67
  
```

Fig 1.10 Corresponding ICMP reply

可以看到捕获到的 ICMP 报文由以下部分组成：

1. Type: 8

类型，占 1 个字节，表示该 ICMP 的类型。使用 ping 时其值为 8 或 0，分别表示 Echo (ping) request 与 Echo (ping) reply。

2. Code: 0

代码，占 1 个字节，用于进一步区分同一个 Type 下的不同情况。此处当 Type 为 8, Code 为 0 时，代表回显 (ping) 请求。

3. Checksum: 0x23d5 [correct]

检验和，占 2 个字节，用于检验整个 ICMP 报文，其计算方法与 IP 首部检验和的计算方法一样。

- 以下四个字节与 ICMP 报文类型相关。

1. Identifier (BE): 29395 (0x72d3)

标识符字段，占 2 个字节，BE 表示是大端模式表示。用于标识本 ICMP 进程。

2. Sequence number (LE): 256 (0x0100)

序列号字段，占 2 个字节，LE 表示是小端模式表示。用于判断回送应答报文。

- 以下为数据部分。

1. Data

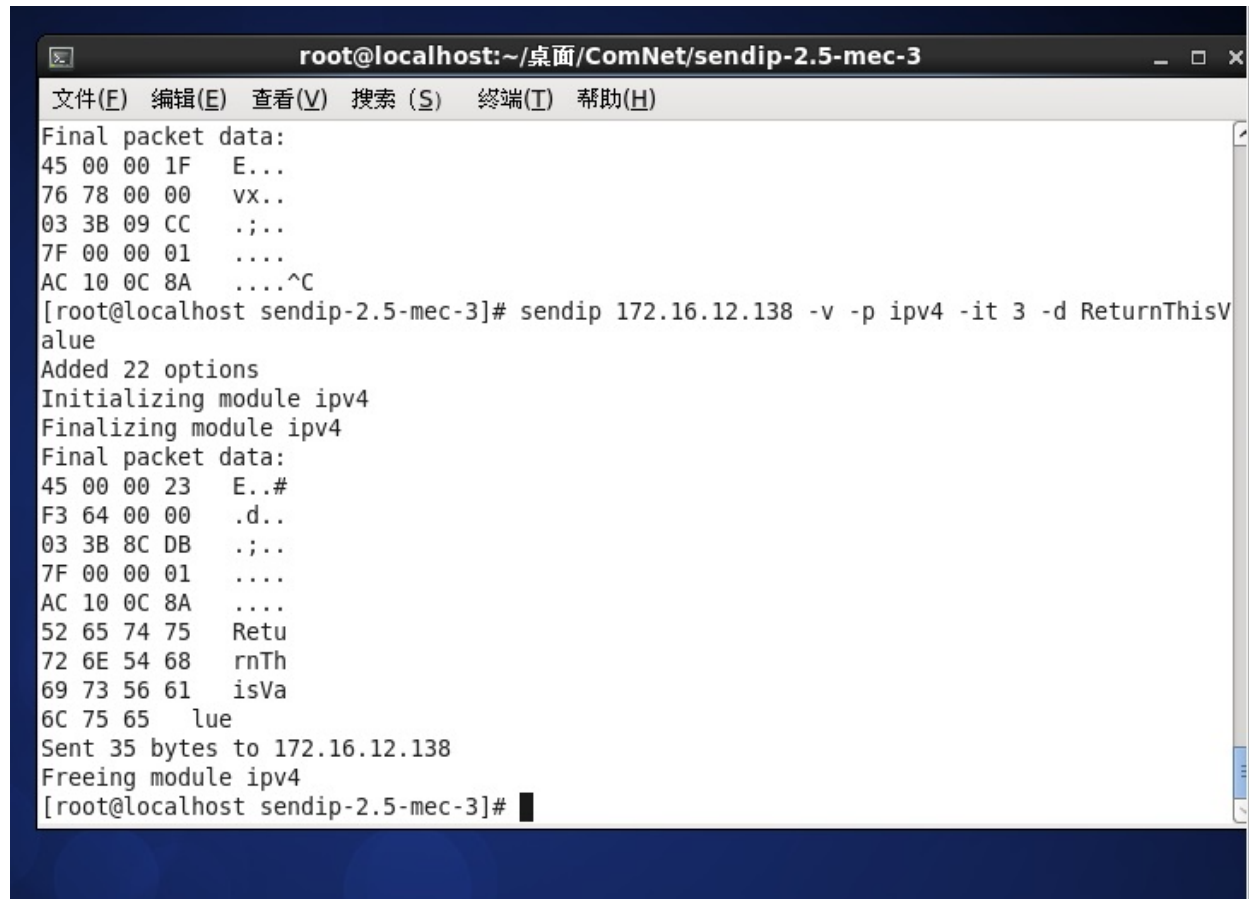
数据部分，长度取决于 ICMP 报文类型，此处为 48 个字节。

- 对于一对由 ping 指令发起的 ICMP 请求，每次 ping 源主机都会向目的 IP 发出一个回送请求报文并记录时间戳，如果目的主机能够正常响应 ping 请求报文则会发回一个回送应答报文，其中回答报文的标识符字段与请求报文一致。源主机收到目的主机的正确应答报文后，通过计算时间戳

差值便容易得出此次 ping 所消耗的时间，完成一次 ping 指令。

实验 2. 使用 SendIP 构造 IP 数据报

本实验中我们将在 CentOS 7 虚拟机环境下使用 SendIP-2.5-mec-3 构造一个使用 IPv4 协议的数据报，并将其数据部分设置为 `ReturnThisValue`，尝试在 Windows 下使用 Wireshark 进行捕获。Windows 与 CentOS 7 虚拟机的网络连接方式为 NAT。实验结果如图 2.1 与 2.2 所示。



```
root@localhost:~/桌面/ComNet/sendip-2.5-mec-3
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Final packet data:
45 00 00 1F  E...
76 78 00 00  vx..
03 3B 09 CC  .;..
7F 00 00 01  ....
AC 10 0C 8A  ....^C
[root@localhost sendip-2.5-mec-3]# sendip 172.16.12.138 -v -p ipv4 -it 3 -d ReturnThisValue
Added 22 options
Initializing module ipv4
Finalizing module ipv4
Final packet data:
45 00 00 23  E..#
F3 64 00 00  .d..
03 3B 8C DB  .;..
7F 00 00 01  ....
AC 10 0C 8A  ....
52 65 74 75  Retu
72 6E 54 68  rnTh
69 73 56 61  isVa
6C 75 65    lue
Sent 35 bytes to 172.16.12.138
Freeing module ipv4
[root@localhost sendip-2.5-mec-3]#
```

Fig 2.1 Using SendIP to send packet to Windows

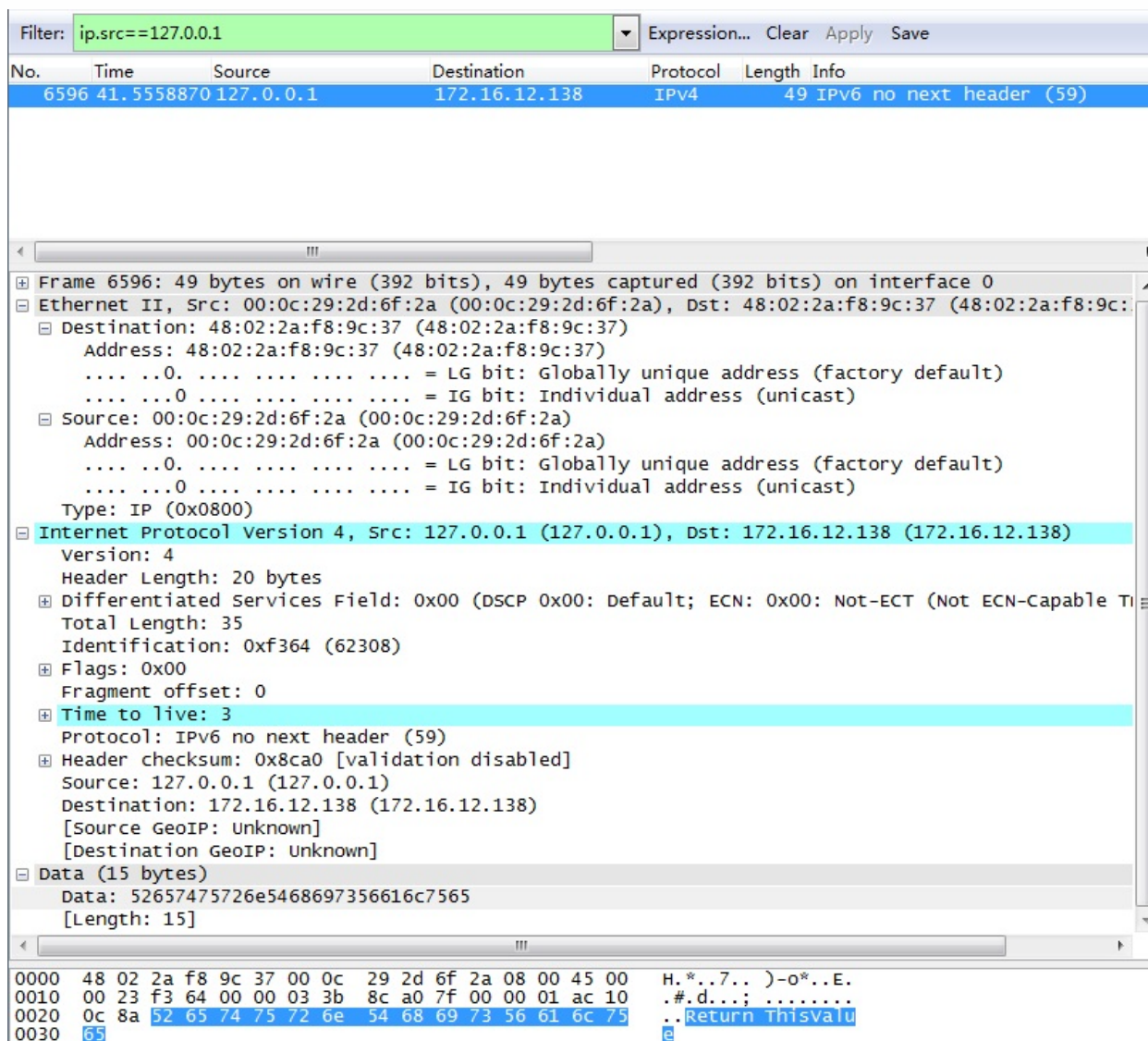


Fig 2.2 Using Wireshark to capture packet sent by sendip

使用命令 `sendip 172.16.12.138 -v -p ipv4 -it 3 -d ReturnThisValue` 来向 Windows 发送数据报，可以看到在 Wireshark 里截获到了数据完全一致的 IP 数据报。

- 问题：使用 SendIP 构造的 ICMP 报文，数据字段最大长度是多少？

- 理论论证：

使用 SendIP 构造的单个 ICMP 报文最大长度受 IP 数据报首部 **总长度** 字段限制，由于总长度字段长为 2 个字节，故最多可接受长为 $2^{16} - 1 = 65535(\text{bytes})$ 的 IP 数据报。但是考虑到数据链路层存在 **最大传输单元 MTU** 的限制，大于 **MTU** 限制（一般为 1500 字节）的 IP 数据报必须进行分片。因而在 MTU 为 1500 字节时，实际上单个 ICMP 报文数据部分大小限制应该在 $1500 - 20 - 8 = 1472(\text{bytes})$ ，其中 20 字节为 IP 数据报首部，8 字节为 ICMP 数据报首部。

下面考虑分片的 ICMP 报文受到的限制。分片的 ICMP 报文受到的限制主要来源于 **片偏移** 字段。片偏移字段指明了当前分片的数据开始位置与整个数据字段起点开始位置的距离，并以 8 个字节为偏移单位，长度为 13 位。故受片偏移字段限制，ICMP 报文数据部分大小限制在 $2^{13} = 8192(\text{bytes})$ 。

综上所述，ICMP 报文数据字段最大长度应该在分片时取得，为 8192 字节。

实验论证：

注意到 sendip 可以通过 -dr[data] 的选项向报文数据部分随机填充 data 字节的数据。

采用指令 `sendip 127.0.0.1 -p ipv4 -p icmp -ct 0 -dr8192`，获得返回结果如图 2.3.

```
47 ED 24 09 G.$.  
ED F6 23 01 ..#.  
CC FB 93 65 ...e  
25 56 B2 2E %V..  
8F 81 2C 0E ...  
F2 14 2F 54 ../T  
00 E8 67 CD ..g.  
FF BD 41 B5 ..A.  
Sent 8216 bytes to 127.0.0.1  
Freeing module ipv4  
Freeing module icmp  
[root@VM_167_127_centos sendip-2.5-mec-3]# sendip 127.0.0.1 -p ipv4 -p icmp -ct 0 -dr8192
```

Fig 2.3 Successfully sent a packet with 8192 bytes data with SendIP

采用指令 `sendip 127.0.0.1 -p ipv4 -p icmp -ct 0 -dr8193`，获得返回结果如图 2.4.

```
[root@VM_167_127_centos sendip-2.5-mec-3]# sendip 127.0.0.1 -p ipv4 -p icmp -ct 0 -dr8193  
Random data too long to be sane
```

Fig 2.4 Failed sending a packet with 8193 bytes data with SendIP

得证以太网下 ICMP 报文数据部分长度最长为 8192 字节。

实验 3. 编程生成 IP 数据报首部

实验要求在程序 输入源 IP 地址与目的 IP 地址 的情况下，以十六进制表示形式按 5 行 * 4 组的排版输出 对应的 20 字节 IP 数据报合法首部。

实验采用 C 编写程序，编译器为 Apple LLVM version 8.0.0, 系统环境为 macOS 10.12.1, 源代码详见附件或本报告末尾 `Source Code` 部分。

程序代码函数 `void printIP8b(void)` (负责按格式打印出十六进制表示的数据报首部), `void fillIPFrag(int *, int)` (负责将输入的 IP 地址填入首部), `void calcChecksum(void)` (负责计算 FCS 校验和) 与主函数构成。IP 首部数据存放在 5 * 8 的无符号 `char` 类型中，每位相当于 4 位，并提前写入了值固定的位。

FCS 校验和的计算方法为二进制求和取反，详见 [RFC 1071](#).

比较 Wireshark 抓取到的 IP 数据报首部 (图 3.1) 与本程序生成的 IP 数据报首部 (图 3.2)，可见两者一致。


```

▼ Internet Protocol Version 4, Src: 52.208.5.180, Dst: 192.168.1.7
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
▼ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    0000 00.. = Differentiated Services Codepoint: Default (0)
    .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
Total Length: 114
Identification: 0xec70 (60528)
▼ Flags: 0x02 (Don't Fragment)
    0... .... = Reserved bit: Not set
    .1.. .... = Don't fragment: Set
    ..0. .... = More fragments: Not set
Fragment offset: 0
Time to live: 49
Protocol: TCP (6)
Header checksum: 0x60e2 [validation disabled]

```

0000	5c 96 9d 7a 25 8d 08 bd 43 d5 08 9e 08 00 45 00	\\.z%... C.....E.
0010	00 72 ec 70 40 00 31 06 60 e2 34 d0 05 b4 c0 a8	.r.p@.1. .4.....
0020	01 07 01 bb db 8d 80 4c 8f 77 f4 d3 00 51 80 18L .w...Q..

Fig 3.1 IP packet header captured by Wireshark

Fig 3.2 Corresponding IP packet header generated by my own implementation

下附程序源代码:

Source Code IP_Head_Generating.c

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

static int frag[4];
static unsigned char iphead4b[5][8] = {{4, 5, 0, 0, 0, 0, 1, 8}, // Init fixed bits
of version/head length/total length
{13, 6, 8, 6, 0}, {15, 15, 0, 1}}; // Init fixed bits of
identification/TTL/protocol number
static short checksum;

void printIP8b(void);
void fillIPFrag(int *, int);
void calcChecksum(void);
void cptChecksum(int, char*);

int
```

```

main(int argc, char *argv[])
{
    printf("Input source IP(xxx.xxx.xxx.xxx): ");
    scanf("%d.%d.%d.%d", &frag[0], &frag[1], &frag[2], &frag[3]);
    fillIPFrag(frag, 0);
    printf("Input destination IP(xxx.xxx.xxx.xxx): ");
    scanf("%d.%d.%d.%d", &frag[0], &frag[1], &frag[2], &frag[3]);
    fillIPFrag(frag, 1);

    calcChecksum();
    printIP8b();

    return 0;
}

void
printIP8b(void)
{
    char p, c, i, j;

    for(i = 0; i < 5; i++){
        for(j = 0; j < 4; j++){
            for(p = 0; p < 2; p++){
                c = iphead4b[i][j * 2 + p];
                if(c < 10 && c >= 0){
                    printf("%c", c + '0');
                }else if(c < 16){
                    printf("%c", c - 10 + 'A');
                }else{
                    printf("\nwrong header value '%d' at [%d][%d]\n", c, i, j);
                    exit(1);
                }
            }
            printf(" ");
        }
        printf("\n");
    }
}

void
fillIPFrag(int* frag, int mode)
{
    int i, j;
    //Locating row to write in
    if(mode != 0){
        mode = 4;
    }else{
        mode = 3;
    }
}

```

```

    }

    //Write in IP Section
    for(i = 0; i < 4; i++){
        j = *(frag+i);

        *(frag+i) = 900; //Set value as PROCESSED for illegal input check
        if(j > 255 || j < 0){
            printf("illegal input\n");
            exit(1);
        }

        iphead4b[mode][i * 2] = j / 16;
        iphead4b[mode][i * 2 + 1] = j % 16;
    }
}

void
calcChecksum(void)
{
    int i, j, sum[4] = {0};
    bool flag;
    for(i = 0; i < 5; ++i){
        for (j = 0; j < 4; ++j)
        {
            sum[j] += iphead4b[i][j] + iphead4b[i][j + 4];
        }
    }
}

/* THE FOLLOWING CHECKSUM CALCULATING METHOD NEEDS TO BE VERIFIED */
while(1){
    flag = true;
    for(i = 3; i >= 0; --i){
        if(sum[i] > 15){
            flag = false;
            if((j = i - 1) < 0){
                j = 3;
            }
            sum[j] += sum[i] / 16;
            sum[i] %= 16;
        }
    }
    if(flag){
        for(i = 0; i < 4; ++i){
            iphead4b[2][i + 4] = 15 - sum[i];
        }
        break;
    }
}

```

```
    }  
  }  
}
```

