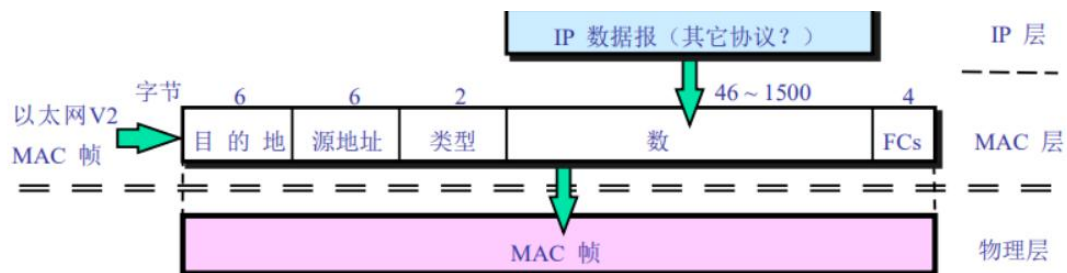


以太网 MAC 帧：

以太网数据帧简称以太帧，起始部分由前同步码和帧开始定界符组成，后面紧跟着一个以太网报头，以 MAC 地址说明目的地址和源地址。以太帧的中部是该帧负载的包含其他协议报头的数据包，最常见的如 IP 协议。另外以太帧由一个 32 位冗余校验码结尾，用于检验数据传输是否出现损坏。

以太网 MAC 帧的地址格式：



前同步码：用来使接收端的网络适配器在接收 MAC 帧时能够迅速调整时钟频率，使它和发送端的频率相同。前同步码为 7 个字节，其值为 1 和 0 交替，即 10101010...

帧开始定界符：帧的起始符，为 1 个字节。其值为：10101011，标志着一个帧的开始，告诉接收端适配器：帧要来了，准备接收。

目的地址：接收帧的网络适配器的物理地址（MAC 地址），为 6 个字节（48 比特）。作用是当网卡接收到一个数据帧时，首先会检查该帧的目的地址，是否与当前适配器的物理地址相同，如果相同，就会进一步处理；如果不同，则直接丢弃。

源地址：发送帧的网络适配器的物理地址（MAC 地址），为 6 个字节（48 比特）。

类型：上层协议的类型，占 2 个字节。由于上层协议众多，所以在处理数据的时候必须设置该字段，标识数据交付哪个协议处理。例如，字段为 0x0800 时，表示将数据交付给网络层的 IP 协议。

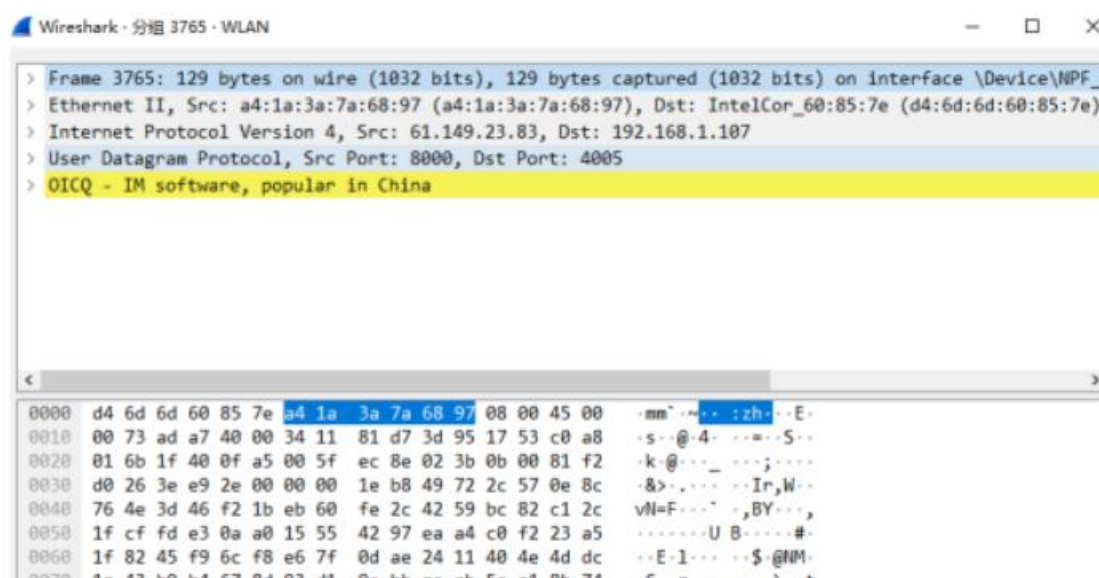
数据：也称为有效载荷，表示交付给上层的数据。以太网帧数据长度最小为 46 字节，最大为 1500 字节，最大值也叫最大传输单元（MTU）。如果不足 46 字节

时，会填充到最小长度。

帧检验序列 FCS：检测该帧是否出现差错，占 4 个字节（32 比特）。发送方计算帧的循环冗余码校验（CRC）值，把这个值写到帧里。接收方计算机重新计算 CRC，与 FCS 字段的值进行比较。如果两个值不相同，则表示传输过程中发生了数据丢失或改变。这时，就需要重新传输这一帧。

因此，以上可知以太网数据帧的整体大小在 64~1518 字节之间（不含前导字段 7 字节和帧起始符 1 字节）。

我们采用 Wireshark 进行抓包分析，找到自己 ip 对应的网卡抓取一些数据包，这里以 QQ 程序传输的数据包为例，截图如下：



对上面进行说明：

Frame 3765: 129 bytes on wire (1032 bits), 129 bytes captured (1032 bits) on interface \Device\, id 0: 这句话的意思就是说数据帧号码 3765，捕获了 129 个字节，也就是 1032 位（一字节等于八位），在 interface 0 上面，也就是在网卡 0 上面（一个机器可能有多个网卡）。

Ethernet II：以太帧的包头，可以清楚地看到里面包含三个信息：Destination, Source, Type 依次对应前面介绍以太帧数据结构的目的地址、源地址、类型，而 type 类型值为 0x0800 表示 IPv4，也就是说它是一个 IP 包。

```

    Ethernet II, Src: a4:1a:3a:7a:68:97 (a4:1a:3a:7a:68:97), Dst: IntelCor_60:85:7e (d4:6d:6d:60:85:7e)
    1 Destination: IntelCor_60:85:7e (d4:6d:6d:60:85:7e)
      Address: IntelCor_60:85:7e (d4:6d:6d:60:85:7e)
      .... 0. .... = LG bit: Globally unique address (factory default)
      .... 0. .... = IG bit: Individual address (unicast)
    2 Source: a4:1a:3a:7a:68:97 (a4:1a:3a:7a:68:97)
      Address: a4:1a:3a:7a:68:97 (a4:1a:3a:7a:68:97)
      .... 0. .... = LG bit: Globally unique address (factory default)
      .... 0. .... = IG bit: Individual address (unicast)

```

Internet Protocol Version 4: 第三个就是 IP 数据包。关于 IP 数据包格式我们以后在进行介绍，这里可以看到，这个 IP 数据包的总长度为 115 字节。

```

    Internet Protocol Version 4, Src: 61.149.23.83, Dst: 192.168.1.107
      0100 .... = Version: 4
      .... 0101 = Header Length: 20 bytes (5)
      > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
        Total Length: 115
      Identification: 0xada7 (44455)
      > Flags: 0x40, Don't fragment
        Fragment Offset: 0
        Time to Live: 52
        Protocol: UDP (17)
        Header Checksum: 0x81d7 [validation disabled]
        [Header checksum status: Unverified]
        Source Address: 61.149.23.83

```

User Datagram Protocol: UDP 数据包。这个 IP 数据包高层传输层协议为 UDP，也就是说 QQ 这个应用程序选择 UDP 作为通信协议，而不是 TCP。关于传输层协议也以后再进行介绍。

```

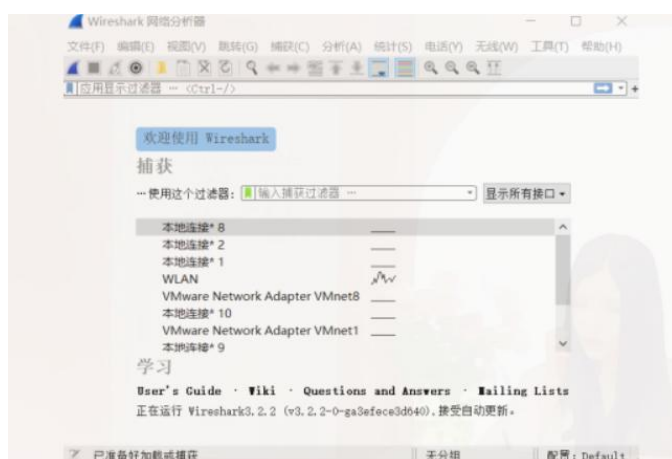
    User Datagram Protocol, Src Port: 8000, Dst Port: 4005
      Source Port: 8000
      Destination Port: 4005
      Length: 95
      Checksum: 0xec8e [unverified]
      [Checksum Status: Unverified]
      [Stream index: 5]
      > [Timestamps]

```

那么，最后我们可以得出捕获到的这个以太帧的总长度为 129 字节，IP 数据包的 total length 是 115 字节，加上以太帧的包头（6+6+2=14）14 字节，115+14 就等于 129 字节，然而，根据前面介绍以太帧尾部还有四个字节的 FCS 校验和。后面的 FCS 四个字节哪里去了呢？是不是我们计算错误了？显然我们的计算是正确的，这是因为数据包经过网络设备，如路由器、交换机等硬件已经把以太帧的校验做过了，它返回给操作系统的只有前面的部分，FCS 没有返回给操作系统，

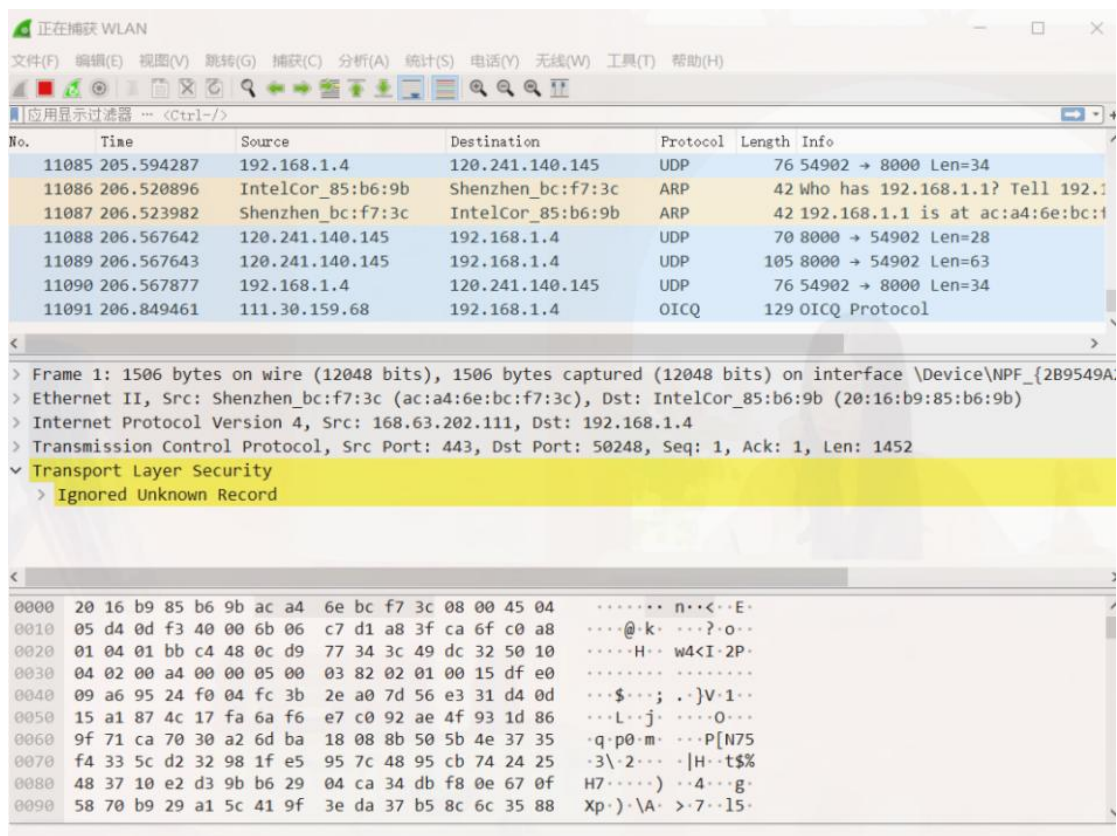
所以 wireshark 也抓不到后面校验的四个字节，当然，校验错误的以太网帧也不会被 wireshark 所捕获，所以，我们看到的以太网帧都是通过校验的正确的以太网帧。

无线局域网 MAC 帧：



打开软件后可以直接在这里选择相应的网卡，也可以在菜单中的捕获->选项，在弹出窗口中选择对应网络接口，这里我选择 WLAN。

选择相应接口后就可以点击开始，开始捕获数据：



顶部是 Packet List Pane（数据包列表窗格），显示了捕获的每个数据包的摘要信息。单击此窗格中的数据包可控制另外两个窗格中显示信息

中间是 Packet Details Pane（数据包详细信息窗格），更加详细的显示了“数据包列表”窗格中所选的数据包。

底部是 Packet Bytes Pane（数据包字节窗格），显示了“数据包列表”窗格中所选数据包的实际数据（以十六进制形式表示实际的二进制），并突出显示了在“数据包详细信息”窗格中所选的字段。

打开 cmd 窗口，ping 网关，然后去软件中抓取相应的包：

```
C:\Users\dell->ping 192.168.1.1

正在 Ping 192.168.1.1 具有 32 字节的数据:
来自 192.168.1.1 的回复: 字节=32 时间=6ms TTL=64
来自 192.168.1.1 的回复: 字节=32 时间=4ms TTL=64
来自 192.168.1.1 的回复: 字节=32 时间=2ms TTL=64
来自 192.168.1.1 的回复: 字节=32 时间=2ms TTL=64

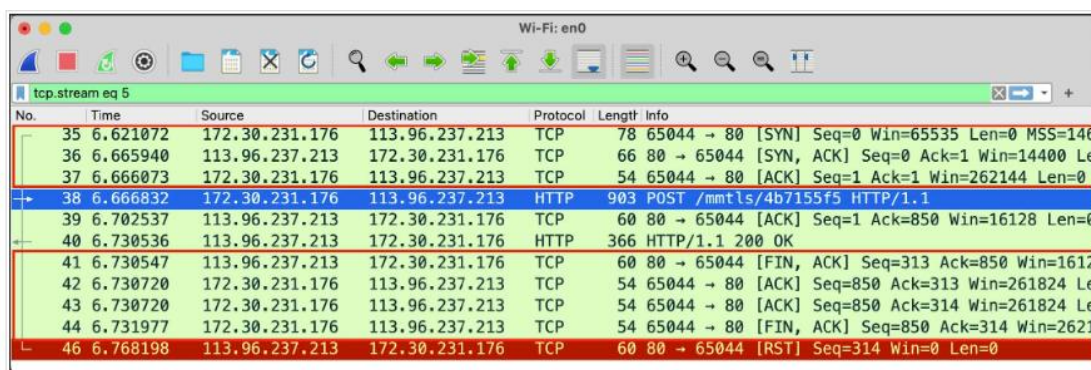
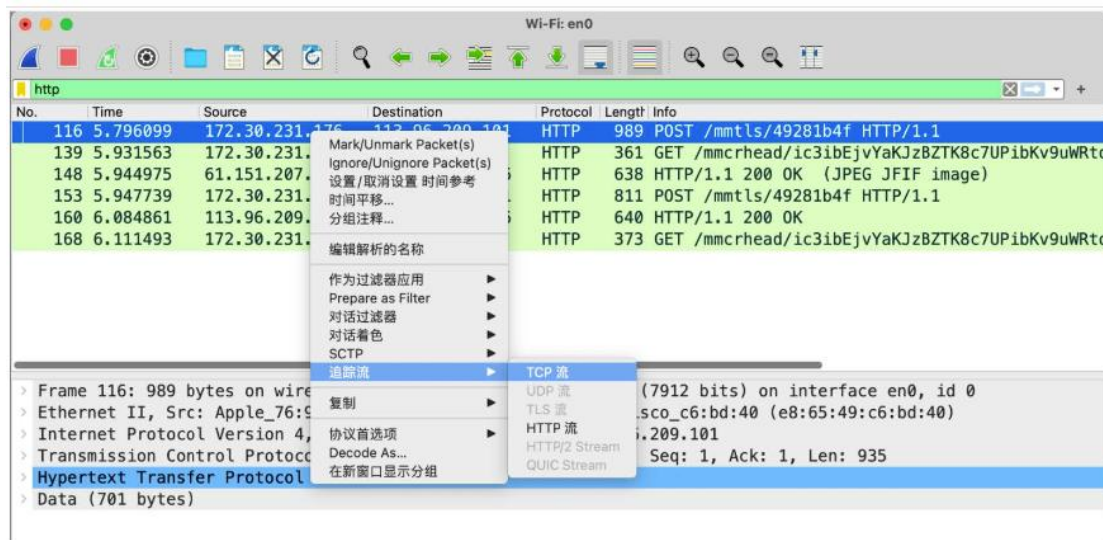
192.168.1.1 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失)
    往返行程的估计时间(以毫秒为单位):
        最短 = 2ms, 最长 = 6ms, 平均 = 3ms
```

TCP 协议：

TCP 被称为是面向连接的(connection oriented)，这是因为在一个应用进程可以开始向另一个应用进程发送数据之前，这两个进程必须互相先“握手”，即它们必须相互发送某些预备报文段，以建立确保数据传输的参数。



图 3-29 TCP 报文段结构



利用 Wireshark 抓取一个 TCP 抓取数据包， 查看其具体数据结构和实际的数据：

```

1  Transmission Control Protocol, Src Port: 50177, Dst Port: 443, Seq: 1, Ack: 2,
2  Source Port: 50177 # 源端口号 C4 01
3  Destination Port: 443 # 目的端口号 01 bb
4  [Stream index: 16]
5  [TCP Segment Len: 0]
6  Sequence Number: 1 (relative sequence number) # 序号 12 2f 1
7  Sequence Number (raw): 305131892
8  [Next Sequence Number: 1 (relative sequence number)]
9  Acknowledgment Number: 2 (relative ack number) # 确认号 60 c2 5
10 Acknowledgment number (raw): 1623348681
11 0101 .... = Header Length: 20 bytes (5) # 首部长度的长8个bit 50
12 Flags: 0x010 (ACK)
13 000. .... = Reserved: Not set # 保留未用 50 10
14 ...0 .... = Nonce: Not set
15 .... 0... = Congestion Window Reduced (CWR): Not set
16 .... .0.. = ECN-Echo: Not set
17 .... .0. .... = Urgent: Not set
18 .... ..1 .... = Acknowledgment: Set # Flag中的ACK已经被标记
19 .... .... 0... = Push: Not set
20 .... .... .0.. = Reset: Not set

```

```

21      .... ..0. = Syn: Not set
22      .... ..0. = Fin: Not set
23      # 其余Flag均未被标记
24      [TCP Flags: .....A...]
25      Window: 4096 # 接收窗口大小 1
26      [Calculated window size: 4096]
27      [Window size scaling factor: -1 (unknown)]
28      Checksum: 0x5abf [unverified] # 因特网检验和 5a bf
29      [Checksum Status: Unverified]
30      Urgent Pointer: 0 # 紧急数据指针 00 00
31      [SEQ/ACK analysis]
32      [Timestamps]

```

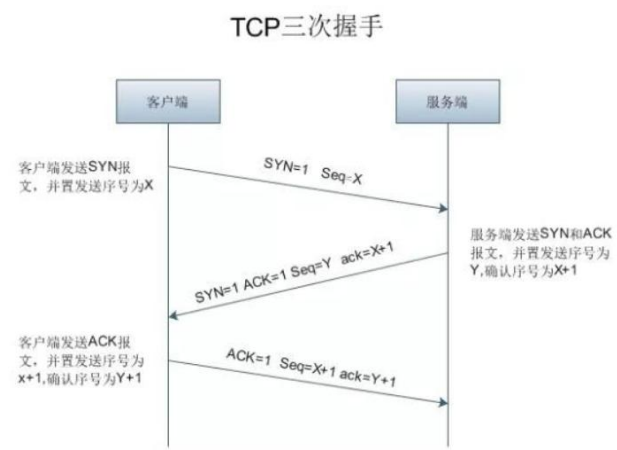
```

Transmission Control Protocol, Src Port: 50177, Dst Port: 443, Seq: 1, Ack: 2, Len: 0
Source Port: 50177
Destination Port: 443
[Stream index: 16]
[TCP Segment Len: 0]
Sequence Number: 1 (relative sequence number)
Sequence Number (raw): 305131892
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 2 (relative ack number)
Acknowledgment number (raw): 1623348681
0101 .... = Header Length: 20 bytes (5)
Flags: 0x010 (ACK)
 000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... = Congestion Window Reduced (CWR): Not set
.... .0.. = ECN-Echo: Not set
.... ..0. = Urgent: Not set
.... ...1 = Acknowledgment: Set
.... .... 0... = Push: Not set

0000 50 d2 f5 2e de 27 4c 20 b8 e2 a1 2e 08 00 45 00  P...L...E.
0010 00 28 00 00 40 00 40 06 75 a7 c0 a8 1f 0f b4 65  .(...@.u.....e
0020 31 0c c4 01 01 bb 12 2f f1 74 60 c2 55 c9 50 10  1...../..t..U.P.
0030 10 00 5a bf 00 00                                ..Z...

```

下面我们说一下 TCP 的三次握手：



No.	Time	Source	Destination	Protocol	Length	Info
1151	36.192597	192.168.31.15	188.101.49.12	TCP	78	50944 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=2945411984 TSecr=0 SACK_PERM=1
1152	36.193236	192.168.31.15	188.101.49.12	TCP	78	50945 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=2682664700 TSecr=0 SACK_PERM=1
1153	36.213609	188.101.49.12	192.168.31.15	TCP	78	443 → 50945 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1440 WS=32 SACK_PERM=1
1154	36.213612	188.101.49.12	192.168.31.15	TCP	78	443 → 50944 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1440 WS=32 SACK_PERM=1
1155	36.213678	192.168.31.15	188.101.49.12	TCP	54	50945 → 443 [ACK] Seq=1 Ack=1 Win=262144 Len=0
1156	36.213678	192.168.31.15	188.101.49.12	TCP	54	50944 → 443 [ACK] Seq=1 Ack=1 Win=262144 Len=0
1157	36.213960	192.168.31.15	188.101.49.12	TLShv1..	571	Client Hello
1158	36.214015	192.168.31.15	188.101.49.12	TLShv1..	571	Client Hello

这就是本地电脑和百度三次握手的过程。每次握手的过程这里重复了两次，但内容是一样的。

第一次握手：首先，是从本地客户端发向百度服务器的一个数据报。在这个数据报中，Seq=0，说明一开始是从序号为0的包好事发送的。我们看到这里 SYN 这一位已经被设为1了，因为还没收到来自百度的确认信息，因此这里 ACK 为设为0。

```

Flags: 0x002 (SYN)
000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... = Congestion Window Reduced (CWR): Not set
.... .0.. = ECN-Echo: Not set
.... ..0. = Urgent: Not set
.... ...0 = Acknowledgment: Not set
.... .... 0... = Push: Not set
.... ..... 0... = Reset: Not set
> .... .... ..1. = Syn: Set
.... .... ...0 = Fin: Not set
[TCP Flags: .....S.]
Window: 65535
  
```

第二次握手：这是百度服务器给本地电脑发送的报文，其中 Seq=0, ACK = 1，因为 TCP 是全双工通信的，因此从百度发送来的第一个报文段也是从 seq=0 开始的。但是这个报文段中包含了对我发给百度的包的确认信息，因此这里 ACK 被设置了，且值为1，这说明1以前的包我全部都收到了，请本地发送1以后

的包。

```
Flags: 0x012 (SYN, ACK)
 000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... = Congestion Window Reduced (CWR): Not set
.... .0.. = ECN-Echo: Not set
.... ..0. = Urgent: Not set
.... ...1 .... = Acknowledgment: Set
.... .... 0... = Push: Not set
.... .... .0.. = Reset: Not set
> .... .... ..1. = Syn: Set
.... .... ...0 = Fin: Not set
[TCP Flags: .....A..S.]
```

第三次握手：第三次握手是本地发送给百度服务器的，此时， Seq=1，说明这是本地发送的第二个包了（第一个包 Seq=0）； ACK = 1 说明已经收到了来自服务端的 1 以前的所有包。

```
Source Port: 50945
Destination Port: 443
[Stream index: 44]
[TCP Segment Len: 0]
Sequence Number: 1 (relative sequence number)
Sequence Number (raw): 4141424518
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 40655800
0101 .... = Header Length: 20 bytes (5)
✓ Flags: 0x010 (ACK)
 000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... = Congestion Window Reduced (CWR): Not set
.... .0.. = ECN-Echo: Not set
.... ..0. = Urgent: Not set
.... ...1 .... = Acknowledgment: Set
.... .... 0... = Push: Not set
.... .... .0.. = Reset: Not set
.... .... ..0. = Syn: Not set
.... .... ...0 = Fin: Not set
[TCP Flags: .....A.....]
```

下面我们说一下 TCP 的四次挥手：

当通信双方完成数据传输，需要进行 TCP 连接的释放，由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送

任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动，一个 TCP 连接在收到一个 FIN 后仍然能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。因为正常关闭过程需要发送 4 个 TCP 帧，因此这个过程也叫四次挥手。

5921	110.667551	180.101.49.12	192.168.31.15	TCP	54	443 → 53010	[FIN, ACK] Seq=80634 Ack=6224 Win=1648 Len=0
5922	110.667771	192.168.31.15	180.101.49.12	TCP	54	53010 → 443	[ACK] Seq=6224 Ack=80635 Win=4096 Len=0
5923	110.673461	192.168.31.15	180.101.49.12	TCP	54	53010 → 443	[FIN, ACK] Seq=6224 Ack=80635 Win=4096 Len=0
5927	110.689579	180.101.49.12	192.168.31.15	TCP	54	443 → 53010	[RST] Seq=80635 Win=0 Len=0
5993	113.694391	180.101.49.12	192.168.31.15	TCP	54	443 → 53010	[RST] Seq=80635 Win=0 Len=0

第一次挥手：第一次挥手是百度向本地发送的一个包，Seq=80634 Ack =6224。同时，设定了 Fin 位，表示服务器单方面想和本地断开联系。

```
Sequence Number: 80634      (relative sequence number)
Sequence Number (raw): 2312777196
[Next Sequence Number: 80635      (relative sequence number)]
Acknowledgment Number: 6224      (relative ack number)
Acknowledgment number (raw): 560901074
0101 .... = Header Length: 20 bytes (5)
Flags: 0x011 (FIN, ACK)
 000. .... = Reserved: Not set
...0 .... = Nonce: Not set
.... 0... = Congestion Window Reduced (CWR): Not set
.... .0.. = ECN-Echo: Not set
.... ..0. = Urgent: Not set
.... ...1 .... = Acknowledgment: Set
.... .... 0... = Push: Not set
.... .... .0.. = Reset: Not set
.... .... ..0. = Syn: Not set
> .... .... ...1 = Fin: Set
[TCP Flags: .....A...F]
```

第二次挥手：第二次挥手是本地收到了百度想要结束的 Fin 之后，返回了一个带有 ACK 的包。同时告诉服务器，ACK=80635, 说明 80635 以前的数据段都已经收到了。

```

Sequence Number: 6224      (relative sequence number)
Sequence Number (raw): 560901074
[Next Sequence Number: 6224      (relative sequence number)]
Acknowledgment Number: 80635      (relative ack number)
Acknowledgment number (raw): 2312777197
0101 .... = Header Length: 20 bytes (5)
Flags: 0x010 (ACK)
  000. .... = Reserved: Not set
  ...0 .... = Nonce: Not set
  .... 0... = Congestion Window Reduced (CWR): Not set
  .... .0.. = ECN-Echo: Not set
  .... ..0. = Urgent: Not set
  .... ...1 = Acknowledgment: Set
  .... .... 0... = Push: Not set
  .... .... .0.. = Reset: Not set
  .... .... ..0. = Syn: Not set
  .... .... ...0 = Fin: Not set
  [TCP Flags: .....A....]
Window: 4096

```

第三次挥手：前两次挥手是主动关闭方断开连接，但是只是单方面关闭连接。现在要被动关闭方来断开连接，才能实现真正的断连。

我们看到这个是 本地向百度发送的包，设置了 Fin 和 ACK，注意到这个包的 ACK 和第二次挥手发送的 ACK 的值是一模一样的，因为在这两次挥手之间没有收到新的来自百度的包。

```

Sequence Number: 6224      (relative sequence number)
Sequence Number (raw): 560901074
[Next Sequence Number: 6225      (relative sequence number)]
Acknowledgment Number: 80635      (relative ack number)
Acknowledgment number (raw): 2312777197
0101 .... = Header Length: 20 bytes (5)
Flags: 0x011 (FIN, ACK)
  000. .... = Reserved: Not set
  ...0 .... = Nonce: Not set
  .... 0... = Congestion Window Reduced (CWR): Not set
  .... .0.. = ECN-Echo: Not set
  .... ..0. = Urgent: Not set
  .... ...1 = Acknowledgment: Set
  .... .... 0... = Push: Not set
  .... .... .0.. = Reset: Not set
  .... .... ..0. = Syn: Not set
> .... .... ...1 = Fin: Set
  [TCP Flags: .....A...F]
Window: 4096

```

第四次挥手：第四次挥手，是百度收到了本地发送的带有 FIN 标志的包后，返回的确认报文。

这个报文比较特殊，因为返回的报文只设置了 RST 位，没有设置 Seq, Ack。RST 位被设置以后，接收端收到之后不必发送 ACK 包。

```
Sequence Number: 80635      (relative sequence number)
Sequence Number (raw): 2312777197
[Next Sequence Number: 80635      (relative sequence number)]
Acknowledgment Number: 0
Acknowledgment number (raw): 0
0101 .... = Header Length: 20 bytes (5)
Flags: 0x004 (RST)
 000. .... = Reserved: Not set
...0 .... = Nonce: Not set
... 0... = Congestion Window Reduced (CWR): Not set
... .0.. = ECN-Echo: Not set
... ..0. = Urgent: Not set
... ...0 = Acknowledgment: Not set
... .... 0... = Push: Not set
> .... .... .1.. = Reset: Set
... ..0. = Syn: Not set
... ...0 = Fin: Not set
[TCP Flags: .....R..]
```

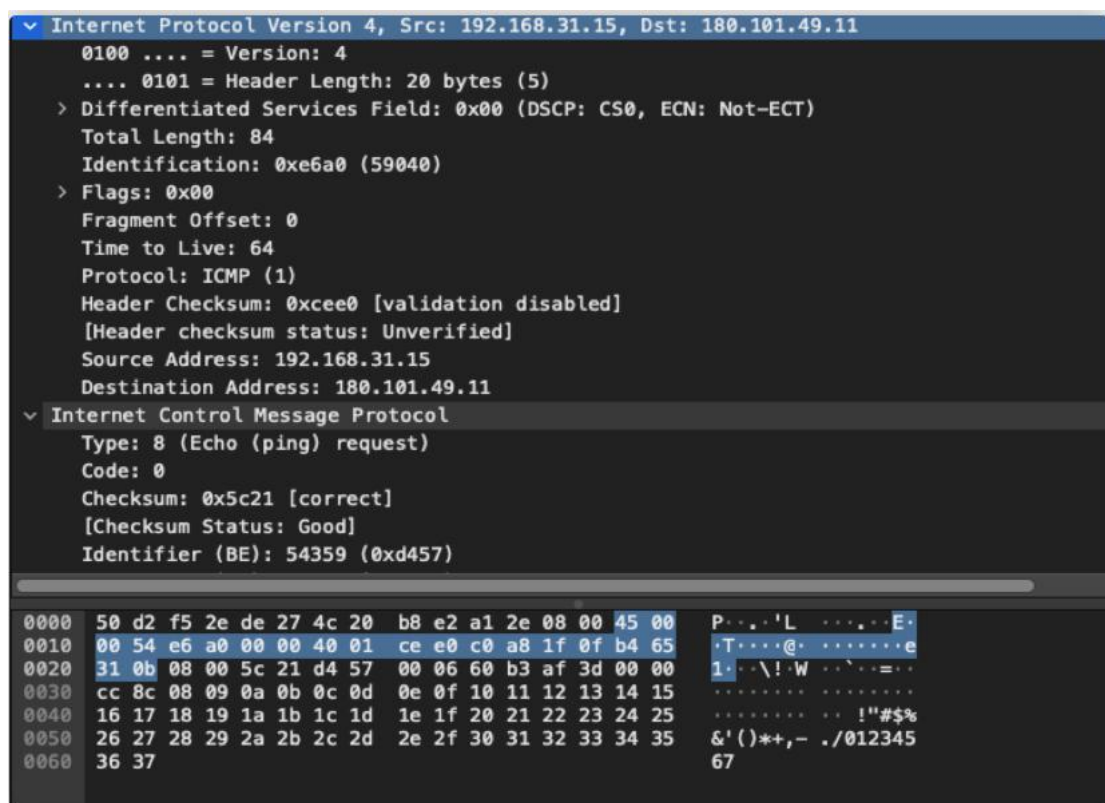
IP 协议：



根据上课学习知道 IP 报文要交给数据链路层封装后才能发送，理想状况下，每个 IP 报文正好能放在同一个物理帧中发送。如果一个数据包超过 1500 字节 (以太网的帧中最多可容纳 1500 字节的数据)，就需要将该包进行分片发送，这个上限被称为物理网络的最大传输单元 (MTU, Maximum Transfer Unit)

TCP/IP 协议在发送 IP 数据报文时，一般选择一个合适的初始长度。当这个报文要从一个 MTU 大的子网发送一个 MTU 小的网络时，IP 协议就把这个报文的数据部分分割成能被目的子网容纳的较小的数据分片，组成较小的报文发送。每个较小的报文被称为一个分片。每个分片都有一个 IP 报文头，分片后的数据包中的 IP 报头和原式 IP 报头分片偏移、MF 标志位和校验字段不同外，其他都一样。

我们取一个有 IP 协议的 ICMP 数据报来分析，并根据该报文分析 IP 协议的报文格式。



```
Internet Protocol Version 4, Src: 192.168.31.15, Dst: 180.101.49.11
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0xe6a0 (59040)
  > Flags: 0x00
    Fragment Offset: 0
    Time to Live: 64
    Protocol: ICMP (1)
    Header Checksum: 0xcee0 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.31.15
    Destination Address: 180.101.49.11
  > Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x5c21 [correct]
    [Checksum Status: Good]
    Identifier (BE): 54359 (0xd457)

0000  50 d2 f5 2e de 27 4c 20 b8 e2 a1 2e 08 00 45 00  P... 'L .....E.
0010  00 54 e6 a0 00 00 40 01 ce e0 c0 a8 1f 0f b4 65  .T....@. ....e
0020  31 0b 08 00 5c 21 d4 57 00 06 60 b3 af 3d 00 00  1... \! W ... =..
0030  cc 8c 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
```

```

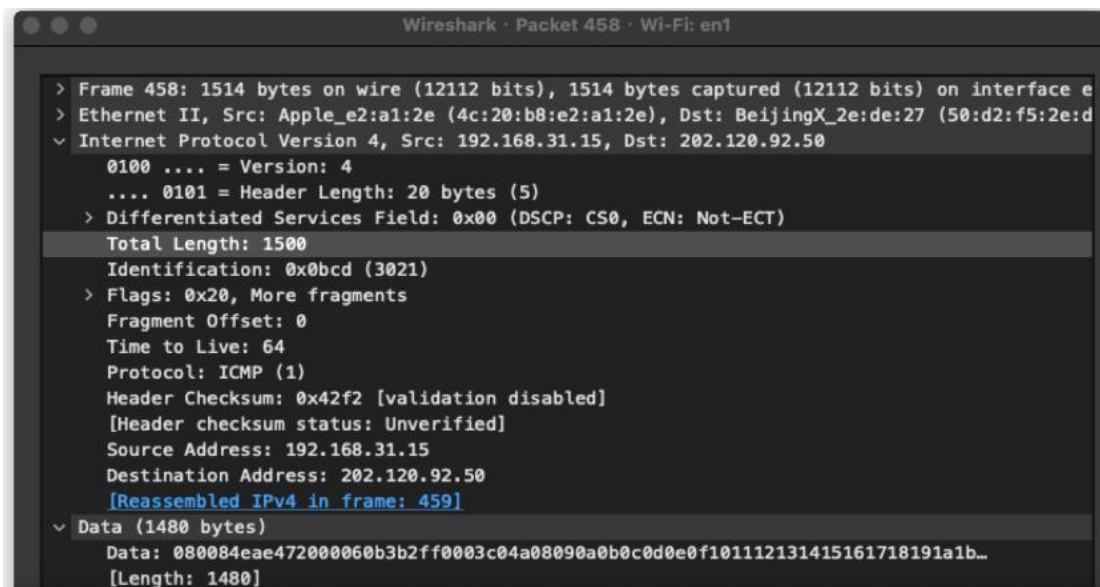
1 Internet Protocol Version 4, Src: 192.168.31.15, Dst: 180.101.49.11 # 源地址和
2 0100 .... = Version: 4 # 版本号, 一般是4
3 .... 0101 = Header Length: 20 bytes (5) # 头部长度, 为20 bytes
4 Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
5 Total Length: 84 #总长度 16进制值为 00 54
6 Identification: 0xe6a0 (59040) # 标志
7 Flags: 0x00 # 标识
8 Fragment Offset: 0 # 片偏移 和flag合起来是 0000
9 Time to Live: 64 # 生存时间 40
10 Protocol: ICMP (1) # 协议 01
11 Header Checksum: 0xcee0 [validation disabled] # 首部检验和 ce e0
12 [Header checksum status: Unverified]
13 Source Address: 192.168.31.15 # 源地址 ce a8 1f 0f
14 Destination Address: 180.101.49.11 # 目标地址 b4 65 31 0b

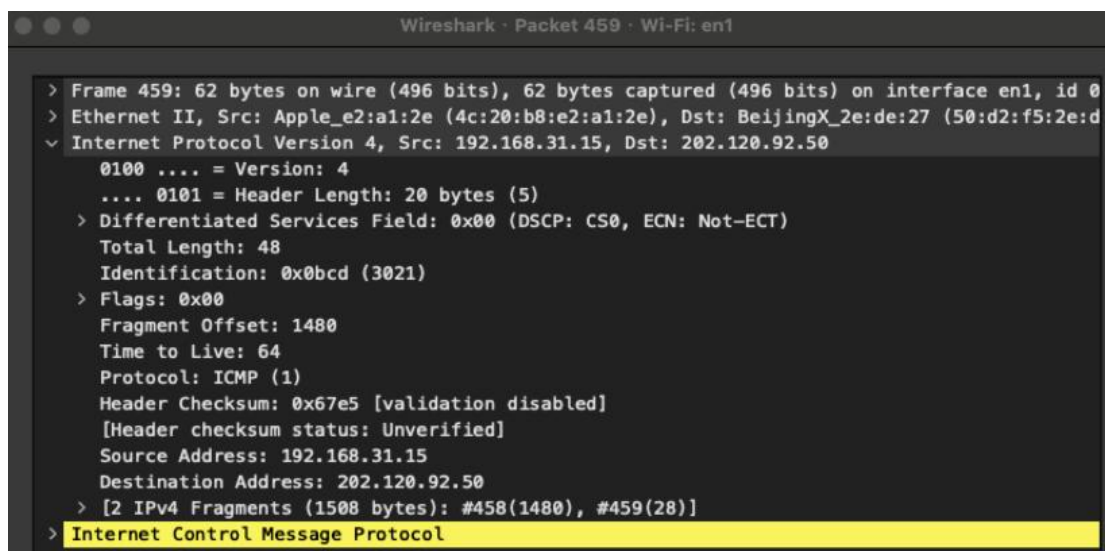
```

我们对截获的报文进行分析，将属于同一个 ICMP 报文的分片找出来，并分析其字节长度特点。

458	12.187110	192.168.31.15	202.120.92.50	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=0, ID=0bcd) [Reassembled in #459]
459	12.187111	192.168.31.15	202.120.92.50	ICMP	62	Echo (ping) request id=0xe472, seq=0/0, ttl=64 (no response found!)
470	13.190146	192.168.31.15	202.120.92.50	IPv4	1514	Fragmented IP protocol (proto=ICMP 1, off=0, ID=2566) [Reassembled in #471]
471	13.190147	192.168.31.15	202.120.92.50	ICMP	62	Echo (ping) request id=0xe472, seq=1/256, ttl=64 (no response found!)

上面产生了两个报文：





我们发现他们的 Identification 都是一样的，为 0x0bcd。由此我们可以断定这两个分片属于同一个 ICMP 报文。

第一段报文是只有 IP 层，没有 ICMP 层的。总长度为 1500，即一个 MTU。但是因为 IP 层头长度为 20，所以 data 的总长度只有 1480。因为这是第一个报文，因此并没有数据偏移量，因此这时 Fragment Offset : 0，同时在 Flags 中也有注明，说有其他分片。

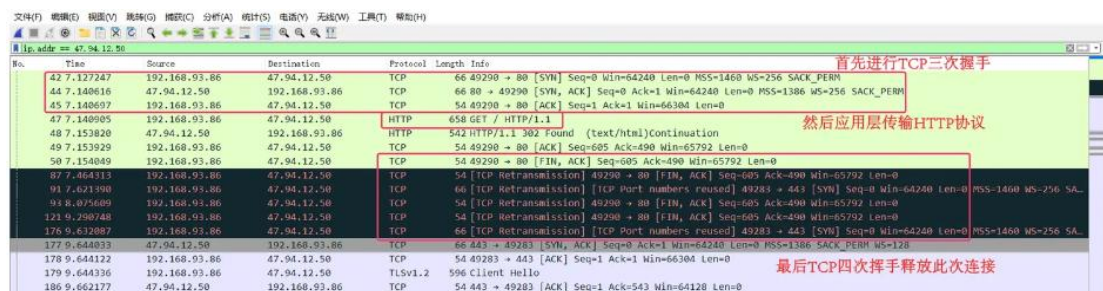
第二段报文报文的总长度为 48，头长度为 20，data 是 28，那么为什么不是 20 而是 28 呢？因为 ICMP 的头部占了 8 个字节。因为这是最后一段报文，因此 Flag 为 0，前面只有一个满的报文，因此 Fragment Offset 为 1480。

因为说不管 ping 的字节有多长，一定都是填满一个报文之后再分出来，所以其 Offset 一定都是 1480 的整数倍。

HTTP 协议：

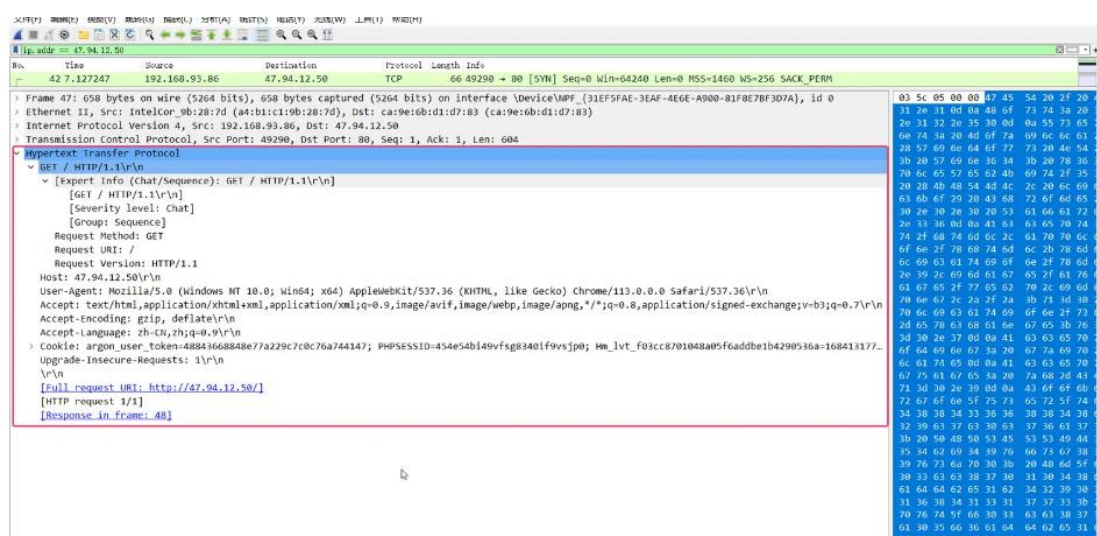
我们首先访问 web 云服务器，以个人网站 <http://47.94.12.50> 为例，后台打开 Wireshark 嗅探，显示过滤器输入 `ip.addr == 47.94.12.50`。

收到的报文如下：



No.	Time	Source	Destination	Protocol	Length	Info
42	7.127247	192.168.93.86	47.94.12.50	TCP	66	49290 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
44	7.140616	47.94.12.50	192.168.93.86	TCP	66	80 → 49290 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1386 WS=256 SACK_PERM
45	7.140697	192.168.93.86	47.94.12.50	TCP	54	49290 → 80 [ACK] Seq=1 Ack=1 Win=66304 Len=0
47	7.140905	192.168.93.86	47.94.12.50	HTTP	658	GET / HTTP/1.1
48	7.153820	47.94.12.50	192.168.93.86	HTTP	542	HTTP/1.1 302 Found (text/html) Continuation
49	7.153929	192.168.93.86	47.94.12.50	TCP	54	49290 → 80 [ACK] Seq=605 Ack=490 Win=65792 Len=0
50	7.154049	192.168.93.86	47.94.12.50	TCP	54	49290 → 80 [FIN, ACK] Seq=605 Ack=490 Win=65792 Len=0
87	7.464313	192.168.93.86	47.94.12.50	TCP	54	[TCP Retransmission] 49290 → 80 [FIN, ACK] Seq=605 Ack=490 Win=65792 Len=0
91	7.621390	192.168.93.86	47.94.12.50	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 49283 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SA...
93	8.075609	192.168.93.86	47.94.12.50	TCP	54	[TCP Retransmission] 49290 → 80 [FIN, ACK] Seq=605 Ack=490 Win=65792 Len=0
121	8.280748	192.168.93.86	47.94.12.50	TCP	54	[TCP Retransmission] 49290 → 80 [FIN, ACK] Seq=605 Ack=490 Win=65792 Len=0
176	9.612887	192.168.93.86	47.94.12.50	TCP	66	[TCP Retransmission] [TCP Port numbers reused] 49283 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SA...
177	9.644833	47.94.12.50	192.168.93.86	TCP	66	443 → 49283 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1386 SACK_PERM WS=128
178	9.644122	192.168.93.86	47.94.12.50	TCP	54	49283 → 443 [ACK] Seq=1 Ack=1 Win=66304 Len=0
179	9.644336	192.168.93.86	47.94.12.50	TLSv1.2	596	Client Hello
186	9.662177	47.94.12.50	192.168.93.86	TCP	54	443 → 49283 [ACK] Seq=1 Ack=543 Win=64120 Len=0

首先主机先与云服务器进行第一次 TCP 连接，完成三次握手，然后应用层使用 HTTP 协议进行传输，最后传输层使用 TCP 四次挥手释放第一次 TCP 连接。下面根据自下而上地分析从 MAC 层协议、IP 协议、TCP 协议一直到 HTTP 协议的过程。这里我们主要看 HTTP 协议：



No.	Time	Source	Destination	Protocol	Length	Info
42	7.127247	192.168.93.86	47.94.12.50	TCP	66	49290 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM

Frame 47: 658 bytes on wire (5264 bits), 658 bytes captured (5264 bits) on interface \Device\NPF... (31EF5FAE-3EAF-4E6E-A900-B1F8E7BF307A), Id 0	Ethernet II, Src: Intelcor 9b:2b:7d (a4:b1:c1:9b:2b:7d), Dst: ca:9e:0b:d1:d7:83 (ca:9e:0b:d1:d7:83)	Internet Protocol Version 4, Src: 192.168.93.86, Dst: 47.94.12.50	Transmission Control Protocol, Src Port: 49290, Dst Port: 80, Seq: 1, Ack: 1, Len: 604	Hypertext Transfer Protocol
GET / HTTP/1.1				
[Expert Info (Chat/Sequence): GET / HTTP/1.1]				
[GET / HTTP/1.1]				
[Severity level: Chat]				
[Group: Sequence]				
Request Method: GET				
Request URI: /				
Request Version: HTTP/1.1				
Host: 47.94.12.50				
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36				
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7				
Accept-Encoding: gzip, deflate				
Accept-Language: zh-CN,zh;q=0.9				
Cookie: argon_user_token=48843668848e77a229c7c0c76a744147; PHPSESSID=454e54b1459fsg8401f9vsjpb; Hm_lvt_f03cc8701048a05f6addbe1b4290536a=16841317...				
Upgrade-Insecure-Requests: 1				
[Full request URI: http://47.94.12.50/]				
[HTTP request 1/1]				
[Response in frame: 48]				

由此 HTTP 报文可知主机通过 HTTP/1.1 协议使用 GET 的请求方法向服务器 资源发起请求。

具体的内容分析如下：

Host: 47.94.12.50。代表了请求资源所属的主机。

User-Agen: Mozilla/5.0(Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/113.0.0.0 Safari/537.36。代表主机使用的 HTTP 协议的客户端类型。

Accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signedexchange;v=b3;q=0.7。代表客户端这边支持任何类型的资源。