哈爾濱工業大學
Harbin Institute of Technology

计算机网络
课程实验报告

<table>
<tr><td>实验名称</td><td colspan="6">利用 Wireshark 进行协议分析</td></tr>
<tr><td>姓名</td><td colspan="2">张儒</td><td>院系</td><td colspan="3">软件工程</td></tr>
<tr><td>班级</td><td colspan="2">2137101</td><td>学号</td><td colspan="3">2021112678</td></tr>
<tr><td>任课教师</td><td colspan="2">李全龙</td><td>指导教师</td><td colspan="3">李全龙</td></tr>
<tr><td>实验地点</td><td colspan="2">格物 207</td><td>实验时间</td><td colspan="3">10 月 30 日</td></tr>
<tr><td rowspan="2">实验课表现</td><td>出勤、表现得分(10)</td><td></td><td rowspan="2">实验报告
得分(40)</td><td rowspan="2"></td><td rowspan="2">实验总分</td><td rowspan="2"></td></tr>
<tr><td>操作结果得分(50)</td><td></td></tr>
<tr><td colspan="7">教师评语</td></tr>
<tr><td colspan="7"></td></tr>
</table>

哈尔滨工业大学计算学部
FACULTY OF COMPUTING, HIT

实验目的：

熟悉并掌握 Wireshark 的基本操作，了解网络协议实体之间进行交互以及报文交换的情况。

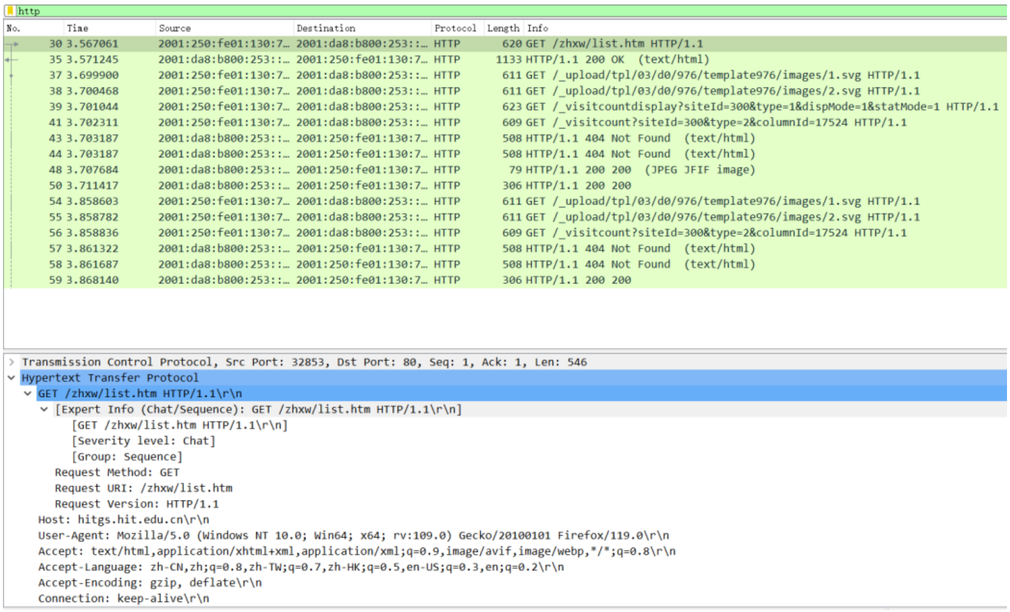
实验内容：

1. 学习Wireshark的基本操作
2. 利用Wireshark分析HTTP协议
3. 利用Wireshark分析TCP协议
4. 利用Wireshark分析IP协议
5. 利用Wireshark分析Ethernet数据帧
6. 利用Wireshark分析DNS协议
7. 利用Wireshark分析UDP协议
8. 利用Wireshark分析ARP协议

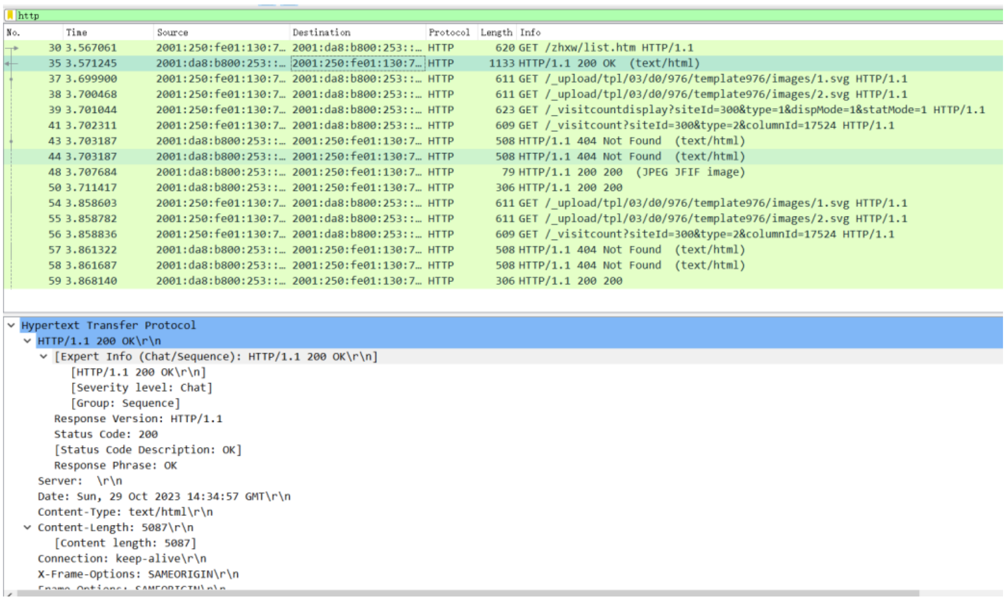
实验过程：

1. 利用Wireshark分析HTTP协议

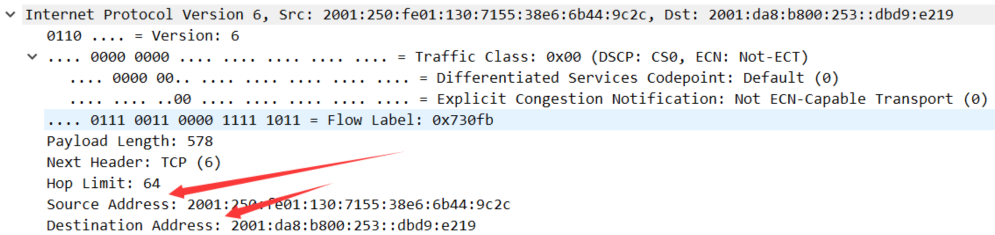
1) 启动浏览器，开始捕获，过滤器输入“http”，访问某个网页，捕获内容如图所示：此图的报文内容为浏览器向服务器的请求报文：



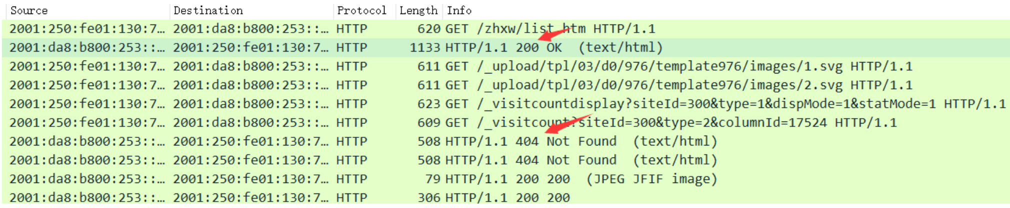
此图的报文内容为服务器返回给客户端的响应报文：



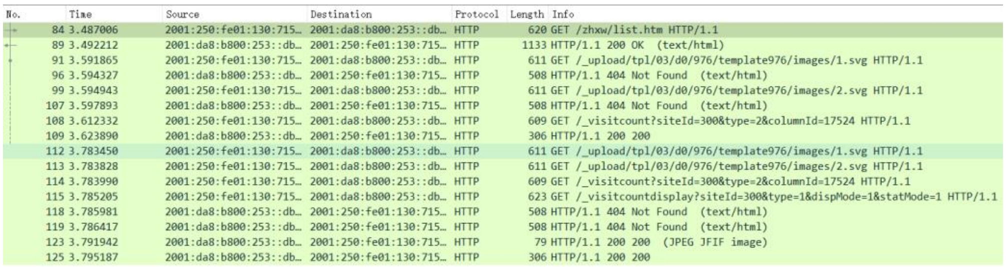
- 可以看到，我的浏览器运行的是HTTP1.1，服务器运行的HTTP协议也为1.1
- 在请求报文的请求头的Accept字段说明了：我的浏览器可以接受：HTML内容（text/html）、XHTML内容(application/xhtml+xml)、XML内容(application/xml)、AVIF格式的图像、WEBP格式的图像。
- 在IPV6协议这一栏，我们可以看到我的计算机的IP以及服务器的IP地址，此处显示的是IPV6地址



- 可以看到，服务器返回的状态码为200和404。访问请求成功以及请求的资源不存在



2) 清空浏览器的缓存，访问网页http://hitgs.hit.edu.cn/zhxw/list.htm，继续捕获



- 可以看到，请求头中并没有If-Modified-Since

```
Hypertext Transfer Protocol
  GET /zhxw/list.htm HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): GET /zhxw/list.htm HTTP/1.1\r\n]
      [GET /zhxw/list.htm HTTP/1.1\r\n]
      [Severity level: Chat]
      [Group: Sequence]
    Request Method: GET
    Request URI: /zhxw/list.htm
    Request Version: HTTP/1.1
  Host: hitgs.hit.edu.cn\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/119.0\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8\r\n
  Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2\r\n
  Accept-Encoding: gzip, deflate\r\n
  Connection: keep-alive\r\n
  Cookie: _ga_Z3B9JJPLKT=GS1.1.1697382027.1.1.1697382074.0.0.0; _ga=GA1.3.34036831.1697382027; JSESSION
    Cookie pair: _ga_Z3B9JJPLKT=GS1.1.1697382027.1.1.1697382074.0.0.0
    Cookie pair: _ga=GA1.3.34036831.1697382027
    Cookie pair: JSESSIONID=150295962387FFFE1A9F2EEB05E78B7C
  Upgrade-Insecure-Requests: 1\r\n
```

之后我又访问了人民网http://www.people.com.cn/，请求头中含有If-Modified-Since字段

```
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): GET / HTTP/1.1\r\n]
      [GET / HTTP/1.1\r\n]
      [Severity level: Chat]
      [Group: Sequence]
    Request Method: GET
    Request URI: /
    Request Version: HTTP/1.1
  Host: www.people.com.cn\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/119.0\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8\r\n
  Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2\r\n
  Accept-Encoding: gzip, deflate\r\n
  Referer: https://www.baidu.com/link?url=LfZiH6UnN5F4-vicgL-vrDrcL6qexSuYuBFu2mWeo67jrqhnoSc2dEpUA_qj09n6
  Connection: keep-alive\r\n
  Cookie: sso_c=0; wdcid=4199fdcd664e0a7e; wdlast=1698590315; wdses=04b6f78a7ed6136e\r\n
    Cookie pair: sso_c=0
    Cookie pair: wdcid=4199fdcd664e0a7e
    Cookie pair: wdlast=1698590315
    Cookie pair: wdses=04b6f78a7ed6136e
  Upgrade-Insecure-Requests: 1\r\n
  If-Modified-Since: Sun, 29 Oct 2023 13:45:54 GMT\r\n
  If-None-Match: W/"653e6212-1ccf4"\r\n
  \r\n
```

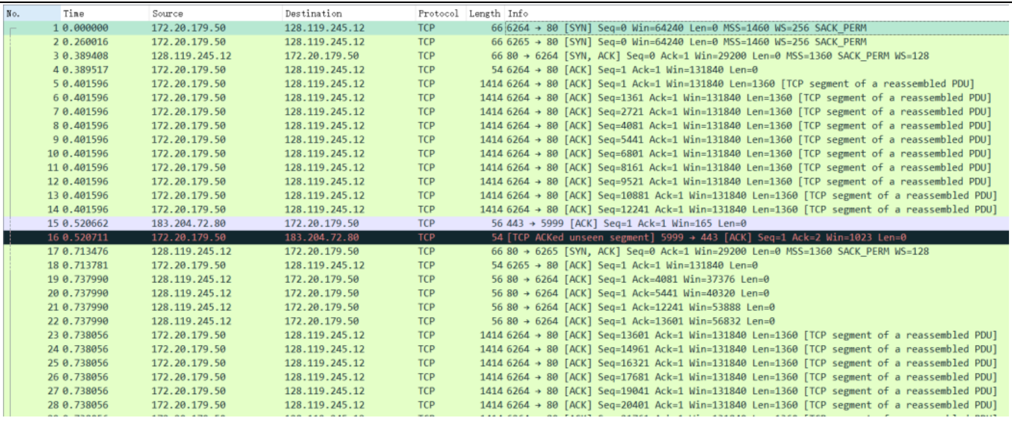
- 在响应报文的Header中的Resuest-URI指明了返回的资源的内容

```
Accept-Ranges: bytes\r\n
Vary: Accept-Encoding\r\n
Content-Encoding: gzip\r\n
X-Frame-Options: SAMEORIGIN\r\n
\r\n
[HTTP response 1/2]
[Time since request: 0.005206000 seconds]
[Request in frame: 84]
[Next request in frame: 91]
[Next response in frame: 96]
[Request URI: http://hitgs.hit.edu.cn/zhxw/list.htm]
Content-encoded entity body (gzip): 5087 bytes -> 20164 bytes
File Data: 20164 bytes
```

2. 利用Wireshark进行TCP协议分析

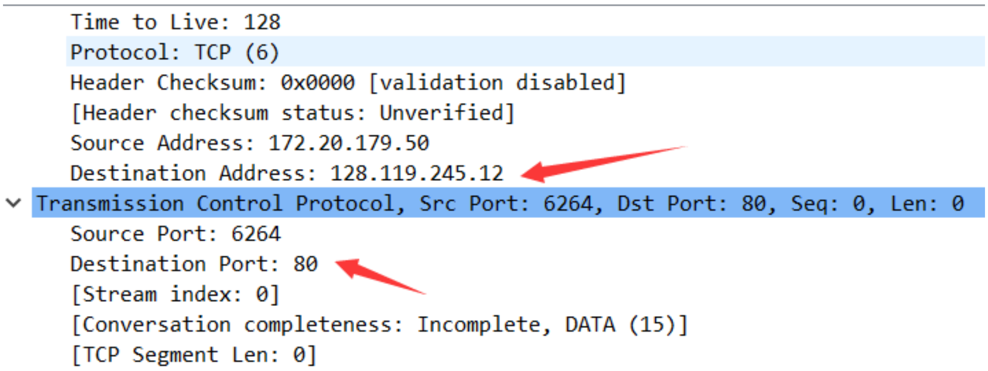
1) 捕获大量的由本地主机到远程服务器的TCP分组

部分分组如图所示：



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.20.179.50	128.119.245.12	TCP	66	6264 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
2	0.260016	172.20.179.50	128.119.245.12	TCP	66	6265 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
3	0.389408	128.119.245.12	172.20.179.50	TCP	66	80 → 6264 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1360 SACK_PERM WS=128
4	0.389517	172.20.179.50	128.119.245.12	TCP	54	6264 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=0
5	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
6	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=1361 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
7	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=2721 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
8	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=4081 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
9	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=5441 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
10	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=6801 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
11	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=8161 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
12	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=9521 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
13	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=10881 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
14	0.401596	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=12241 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
15	0.520662	183.204.72.80	172.20.179.50	TCP	56	443 → 5999 [ACK] Seq=1 Ack=1 Win=165 Len=0
16	0.520711	172.20.179.50	183.204.72.80	TCP	54	[TCP ACKed unseen segment] 5999 → 443 [ACK] Seq=1 Ack=2 Win=1023 Len=0
17	0.713476	128.119.245.12	172.20.179.50	TCP	66	80 → 6265 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1360 SACK_PERM WS=128
18	0.713781	172.20.179.50	128.119.245.12	TCP	54	6265 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=0
19	0.737990	128.119.245.12	172.20.179.50	TCP	56	80 → 6264 [ACK] Seq=1 Ack=4081 Win=37376 Len=0
20	0.737990	128.119.245.12	172.20.179.50	TCP	56	80 → 6264 [ACK] Seq=1 Ack=5441 Win=40320 Len=0
21	0.737990	128.119.245.12	172.20.179.50	TCP	56	80 → 6264 [ACK] Seq=1 Ack=12241 Win=53888 Len=0
22	0.737990	128.119.245.12	172.20.179.50	TCP	56	80 → 6264 [ACK] Seq=1 Ack=13601 Win=56832 Len=0
23	0.738056	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=13601 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
24	0.738056	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=14961 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
25	0.738056	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=16321 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
26	0.738056	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=17681 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
27	0.738056	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=19041 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
28	0.738056	172.20.179.50	128.119.245.12	TCP	1414	6264 → 80 [ACK] Seq=20401 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]

具体报文信息为：

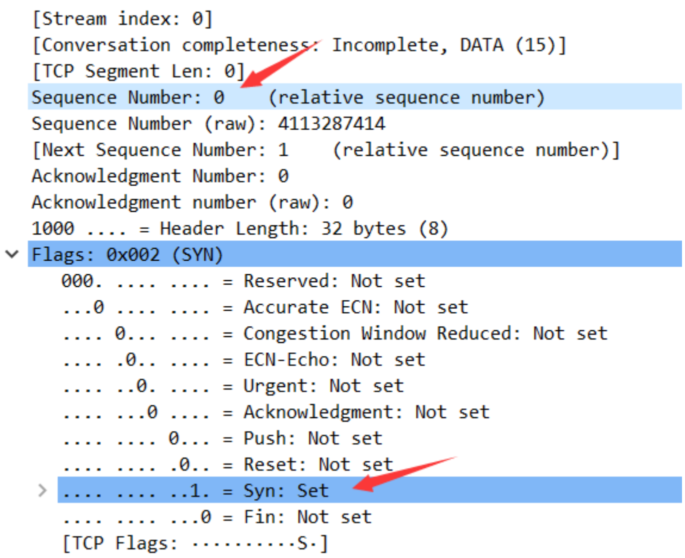


```
    Time to Live: 128
    Protocol: TCP (6)
    Header Checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 172.20.179.50
    Destination Address: 128.119.245.12
v Transmission Control Protocol, Src Port: 6264, Dst Port: 80, Seq: 0, Len: 0
    Source Port: 6264
    Destination Port: 80
    [Stream index: 0]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
```

- 可以看到，向gaia.cs.umass.edu服务器传送文件的客户端主机的IP地址为：172.20.179.50，TCP端口为：6264。Gaia.cs.umass.edu服务器的IP地址为：128.119.245.12，对于这一连接，接受和发送TCP报文的端口号为80

2) 通过捕获的报文对TCP基础进行学习

- 下图为客户服务器之间用于初始化TCP连接的TCP SYN报文段，可以看到，序号seq为0，在TCP请求头的标志为中，将SYN标志位置为了1，用来标识该报文段为SYN报文段。



```
[Stream index: 0]
[Conversation completeness: Incomplete, DATA (15)]
[TCP Segment Len: 0]
Sequence Number: 0    (relative sequence number)
Sequence Number (raw): 4113287414
[Next Sequence Number: 1    (relative sequence number)]
Acknowledgment Number: 0
Acknowledgment number (raw): 0
1000 .... = Header Length: 32 bytes (8)
v Flags: 0x002 (SYN)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0... .... = Congestion Window Reduced: 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·]
```

- 下图为服务器向客户端发送的SYNACK报文：

```
[Stream index: 0]
[Conversation completeness: Incomplete, DATA (15)]
[TCP Segment Len: 0]
Sequence Number: 0    (relative sequence number)
Sequence Number (raw): 1696672062
[Next Sequence Number: 1    (relative sequence number)]
Acknowledgment Number: 1    (relative ack number)
Acknowledgment number (raw): 4113287415
1000 .... = Header Length: 32 bytes (8)
Flags: 0x012 (SYN, ACK)
   000. .... .... = Reserved: Not set
   ...0 .... .... = Accurate ECN: Not set
   .... 0... .... = Congestion Window Reduced: 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
```

- 可以看到，序号seq为0，acknowledgement字段的值为1。
- 服务器收到客户端发来的SYN报文之后，会将其seq + 1 作为返回的SYNACK报文的确认号。
- 可以看到，通过将标志位中的ACK、SYN标志位置为1来标识为SYNACK报文段的。

3) 分析TCP三次握手过程

- 第一次握手：客户端向服务器发送一个SYN报文，等待服务器确认

Source	Destination	Protocol	Length	Info
172.20.179.50	128.119.245.12	TCP	66	6264 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
172.20.179.50	128.119.245.12	TCP	66	6265 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM

- 服务器收到客户端的SYN报文之后，对该报文进行确认，并返回一个SYNACK报文。

```
v Transmission Control Protocol, Src Port: 80, Dst Port: 6264, Seq: 0, Ack: 1, Len: 0
    Source Port: 80
    Destination Port: 6264
    [Stream index: 0]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 1696672062
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 4113287415
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x012 (SYN, ACK)
```

可以看到，SYN、ACK标志位被置为了1，seq为0，ACK序号为客户端的seqx＝0加上1，即为1

- 客户端收到SYN ACK报文之后，确认客户端到服务器的数据传输是正常的，并返回最后一个确认报文。

```
v Transmission Control Protocol, Src Port: 6264, Dst Port: 80, Seq: 1, Ack: 1, Len: 0
    Source Port: 6264
    Destination Port: 80
    [Stream index: 0]
    [Conversation completeness: Incomplete, DATA (15)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 4113287415
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 1696672063
    0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x010 (ACK)
```

可以看到ACK标志位被置为1，seq为seqx加1，即为1。ack序号为seqy加1，即为1。

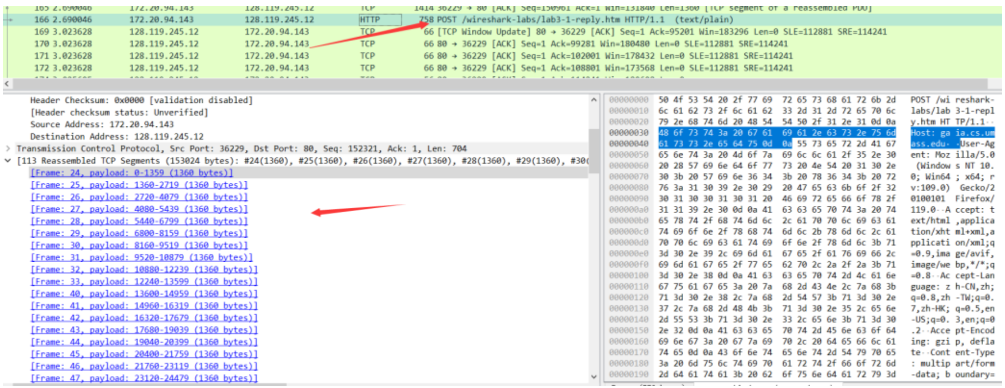
- 最后服务器收到客户端发送的确认报文之后，确认服务器到客户端的数据传输正常的，从而完成三次握手

```
Length Info
  66 6264 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
  66 6265 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
  66 80 → 6264 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1360 SACK_PERM WS=128
  54 6264 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=0
1414 6264 → 80 [ACK] Seq=1 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=1361 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=2721 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=4081 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=5441 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=6801 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=8161 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=9521 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=10881 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
1414 6264 → 80 [ACK] Seq=12241 Ack=1 Win=131840 Len=1360 [TCP segment of a reassembled PDU]
```

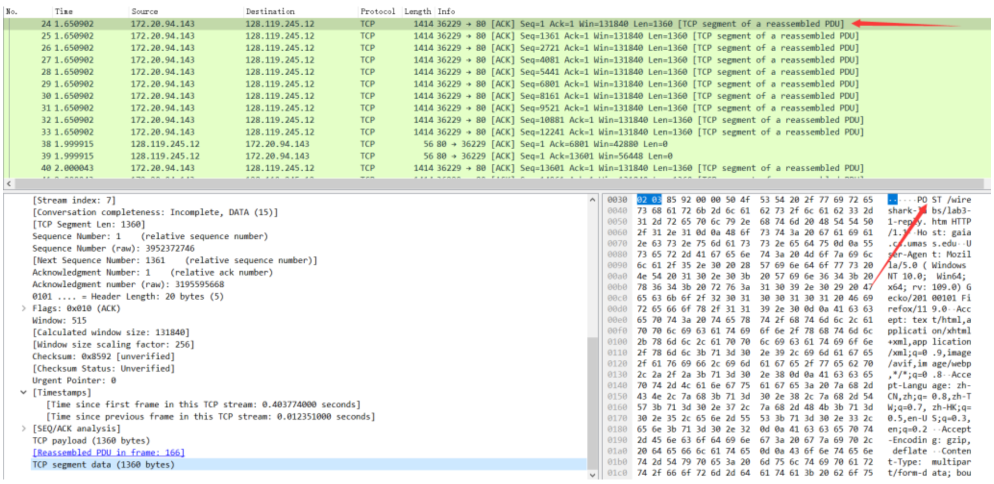
可以看到，握手完成之后，开始了数据传输。

4) 将包含HTTP POST命令的TCP报文段看做是TCP连接上第一个报文段

如果要发送POST请求，会先发送请求行和请求头，再发送请求体。在wireshark中，当POST请求发送完毕之后，会对分段的请求进行一个汇总，也就是下图的请求汇总：

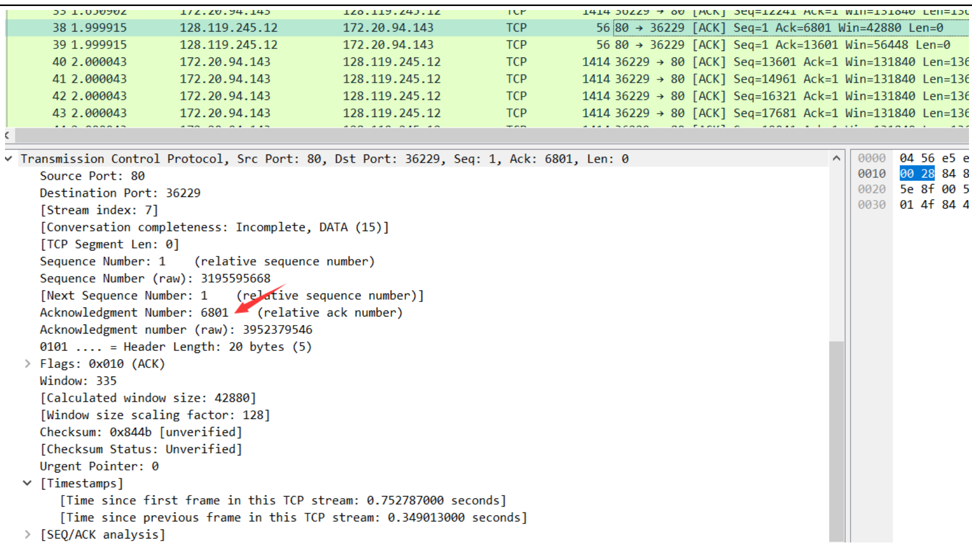


通过汇总，我们可以很容易的找到第一个包含HTTP POST指令的请求头报文段：

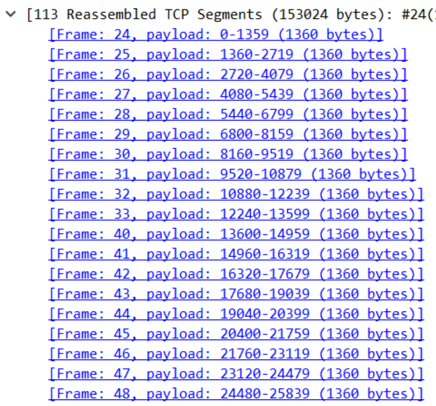


可以看到，在这个TCP报文段的数据里面，有我们的HTTP POST命令。

- 那么第6个报文段的序号为：6801，是当客户端发出POST请求行、请求体之后作为请求体发出的，因此序号为6801，那么对应的ACK为:

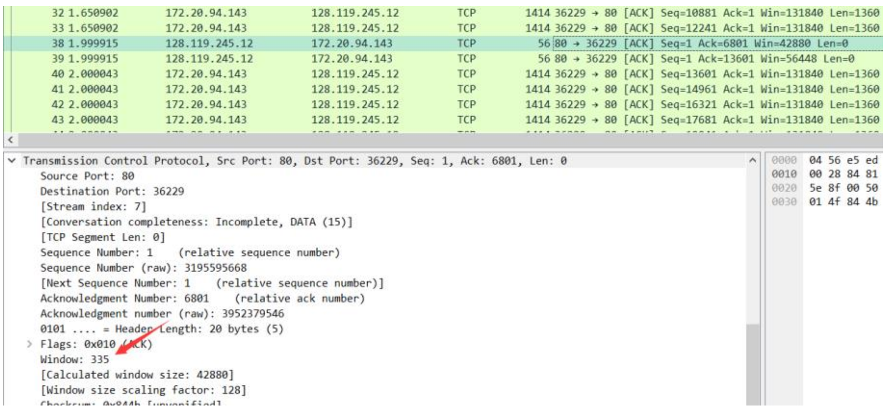


- 通过wireshark对请求的汇总可以看出，前6个TCP报文段的长度均为1360字节



均达到了前面三次握手进行协商的MSS=1360字节。这里我查阅资料显示：POST请求的第一个TCP报文段只是发送请求行和请求头，一般不会携带请求体的内容。所以报文段长度可以不达到MSS，但是我自己实践的时候，第一个TCP段也携带了一些请求体中的数据。也达到了最大长度MSS

- 接收端公示的最小的可用缓存空间为：335。

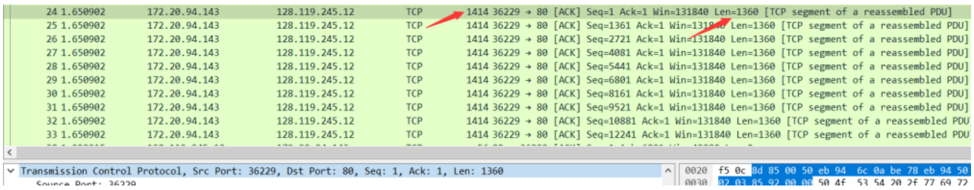


限制发送端的传输以后，接收端的缓存是够用的，接收端公示的最小可用缓存空间再不断增加。

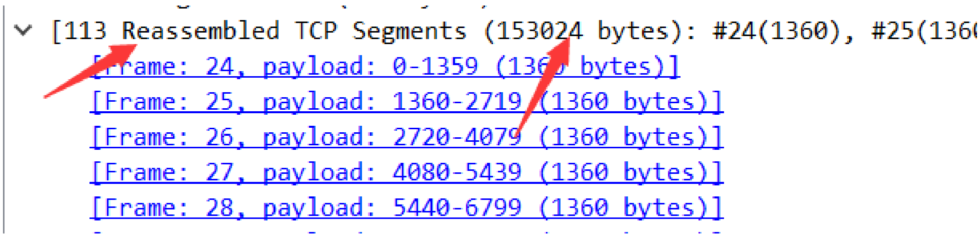
- 并没有重传的报文段，因为所有的TCP的seq都是不相同的。
- Throughput大约为：57.69KB/s

可以看到，一个报文段总长度为1414字节，数据有1360字节，那么TCP头部为54个字

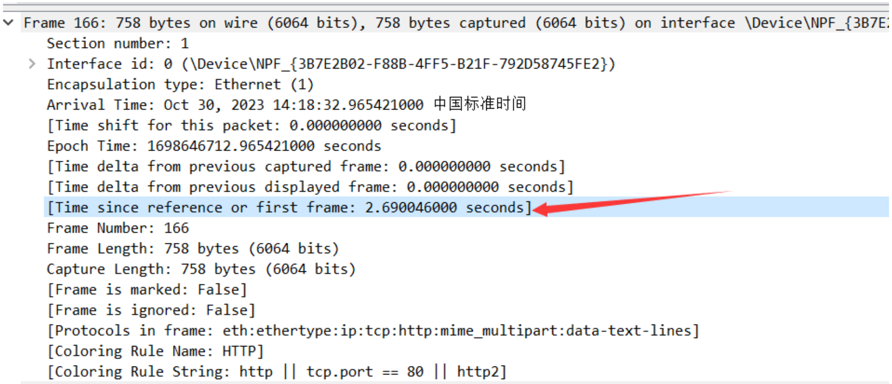
节。



在汇总中我们看到：一共发送了113个TCP段，数据大小为153024字节，那么一共有113*54 + 153024 = 159126字节



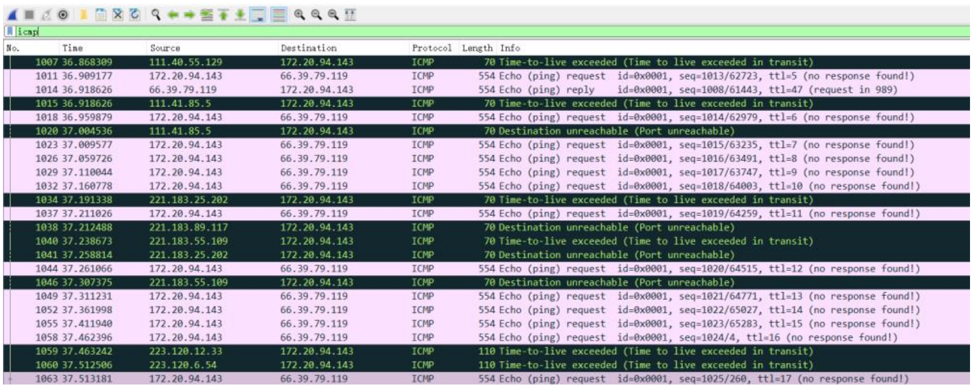
在最后一个TCP段中我们可以看到总时间为：2.690046s



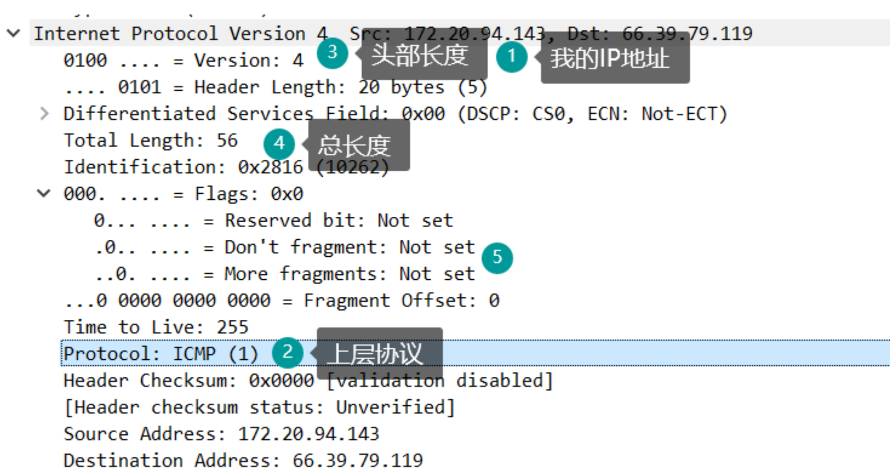
那么Throughput = 159126bytes / 2.690046s = 57.69KB/s

3. 利用Wireshark进行IP协议分析

- 捕获的数据包如图所示：

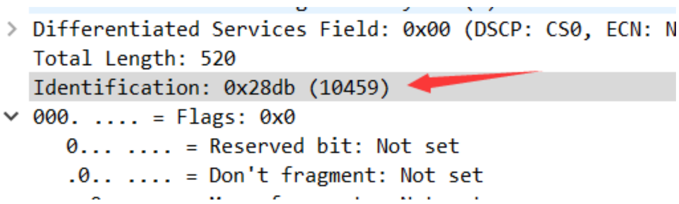


- 我的主机的IP为：172.20.94.143，IP数据包头部，上层协议字段的值为ICMP（1）可以看到IP头有20字节，IP数据包一共是56字节，那么净载就为36字节。通过查看Flags标志位，可以看到More fragments为not set，说明没有分片。

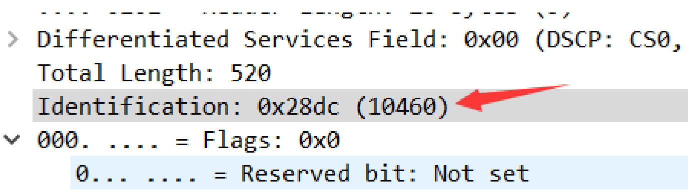


```
Internet Protocol Version 4, Src: 172.20.94.143, Dst: 66.39.79.119
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 56
  Identification: 0x2816 (10262)
  000. .... = Flags: 0x0
    0... .... = Reserved bit: Not set
    .0.. .... = Don't fragment: Not set
    ..0. .... = More fragments: Not set
  ...0 0000 0000 0000 = Fragment Offset: 0
  Time to Live: 255
  Protocol: ICMP (1)
  Header Checksum: 0x0000 [validation disabled]
  [Header checksum status: Unverified]
  Source Address: 172.20.94.143
  Destination Address: 66.39.79.119
```


- 通过分析多个ICMP数据包，我发现:Identification和Time to Live（TTL）字段总是在变，因为需要通过Identification来鉴别不同的数据包，设置不同的TTL是为了检查每一跳的状况。
  其中Total Length也会随着我们发送不同长度的数据包而改变。不同数据大小的数据包的Flags中的Fragment offset也是不同的。其余的字段都为常量。
  注意，如果Header Checksum没有被禁止的话，也是需要改变的，我的主机开启了本地网卡校验和功能，所以本地发出去的包会填充为0，然后交给网卡硬件计算并修改。
- 我看到的Identification字段的形式为16进制，以1为单位递增

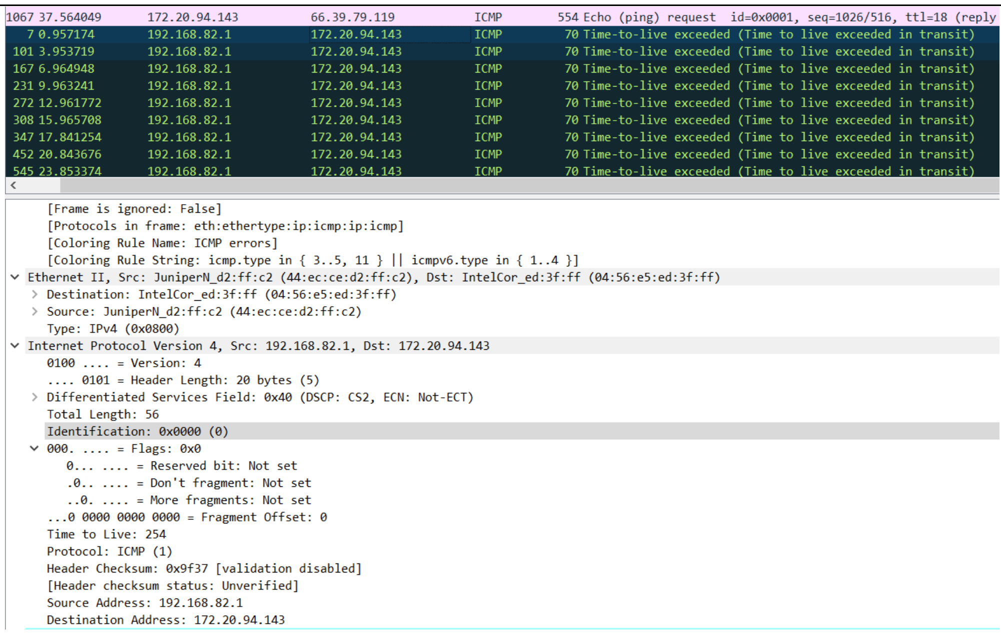


```
Differentiated Services Field: 0x00 (DSCP: CS0, ECN: N
Total Length: 520
Identification: 0x28db (10459)
000. .... = Flags: 0x0
  0... .... = Reserved bit: Not set
  .0.. .... = Don't fragment: Not set
```



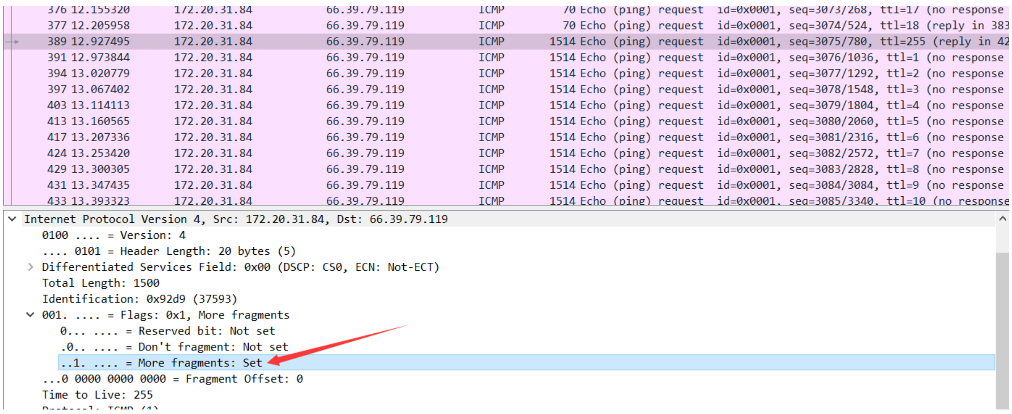
```
Differentiated Services Field: 0x00 (DSCP: CS0,
Total Length: 520
Identification: 0x28dc (10460)
000. .... = Flags: 0x0
  0... .... = Reserved bit: Not set
```

3) 找到由最近的路由器（第一跳）返回给我的主机的ICMP Time-to-live-exceeded消息



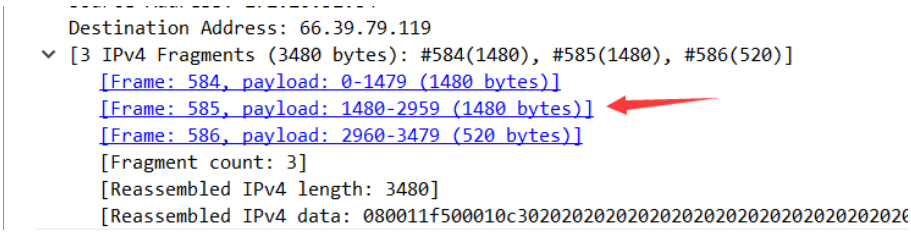
- 可以看到，Identification字段的值为0x0000，TTL字段的值为254
- 最近的路由器返回给我主机的消息中这些值都是不变的，因为

4) 找到改为2000字节后我的主机发送的第一个ICMP Echo Request消息



- 可以看到Flags中More fragments字段为1，说明进行了分片
- IP头部的Flags的标志位的第二位为MF(More fragment)，如果置为1说明后面还有分片，如果为0说明已经是最后一个分片。该分片的长度为1500

5) 找到将包大小改为3500后主机发送的第一个ICMP Echo Request消息



- 可以看到原始数据包被分成了3片
- 这些分片中IP数据报头部的Identification以及TTL发生了变化

4. 利用Wireshark进行ARP协议分析

1) 利用命令arp -a 查看主机ARP缓存内容，结果如下：

```
接口: 192.168.214.1 --- 0x6
  Internet 地址         物理地址              类型
  192.168.214.254       00-50-56-f3-61-2d     动态
  192.168.214.255       ff-ff-ff-ff-ff-ff     静态
  224.0.0.22            01-00-5e-00-00-16     静态
  224.0.0.251           01-00-5e-00-00-fb     静态
  224.0.0.252           01-00-5e-00-00-fc     静态
  239.255.255.250       01-00-5e-7f-ff-fa     静态
  255.255.255.255       ff-ff-ff-ff-ff-ff     静态

接口: 172.20.31.84 --- 0x8
  Internet 地址         物理地址              类型
  172.20.0.1            44-ec-ce-d2-ff-c2     动态
  172.20.0.61           44-ec-ce-d2-ff-c2     动态
  172.20.79.251         44-ec-ce-d2-ff-c2     动态
  172.20.177.221        44-ec-ce-d2-ff-c2     动态
  172.20.183.181        44-ec-ce-d2-ff-c2     动态
  172.20.209.39         44-ec-ce-d2-ff-c2     动态
  172.20.255.255        ff-ff-ff-ff-ff-ff     静态
  224.0.0.22            01-00-5e-00-00-16     静态
  224.0.0.251           01-00-5e-00-00-fb     静态
  224.0.0.252           01-00-5e-00-00-fc     静态
  239.255.255.250       01-00-5e-7f-ff-fa     静态
  255.255.255.255       ff-ff-ff-ff-ff-ff     静态
```

第一列为ip地址，是与本地通信的其他设备，包括路由器、交换机、其他主机等。第二列物理地址为与第一列IP地址相对应的MAC地址，ARP缓存表用于将IP地址映射到对应的MAC地址，以便发送数据包到目标设备。第三列为此映射信息的类型，分为静态和动态。静态ARP条目是手动配置的，动态ARP条目是系统自动学习和更新的。

2) 清空ARP缓存，抓取ping命令时的数据包

ARP

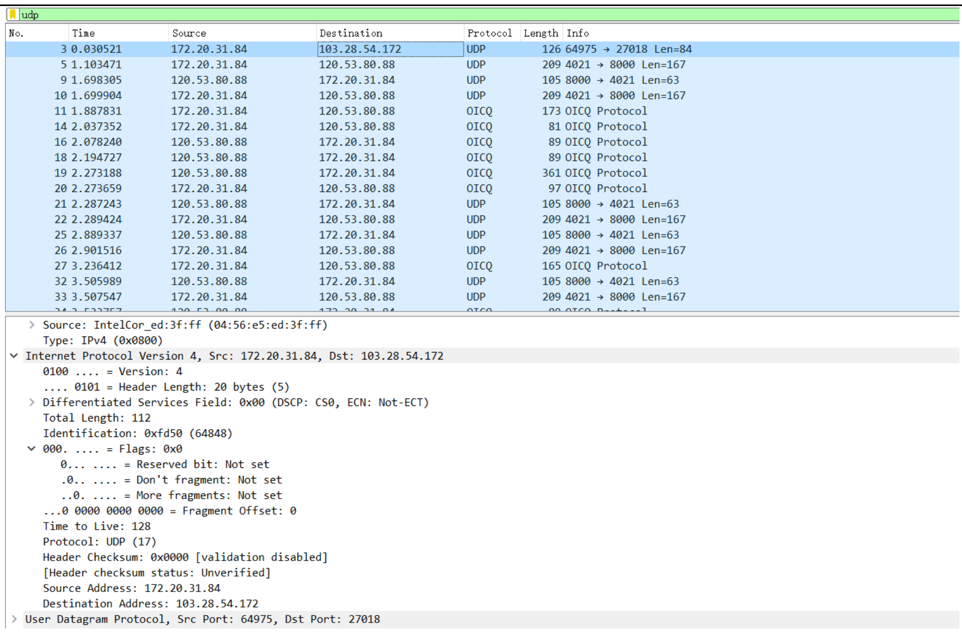
```
v Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: IntelCor_ed:3f:ff (04:56:e5:ed:3f:ff)
    Sender IP address: 172.20.31.84
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 172.20.79.246
```

- 可以看到ARP数据包主要组成有：硬件类型（2字节）、协议类型（2字节）、硬件地址长度（1字节）、协议地址长度（1字节）、操作代码（2字节）、源MAC地址（6字节）、源IP地址（4字节）、目的MAC地址（6字节）、目的IP地址（4字节）
- 通过Opcode操作码字段，我们可以判断是请求包还是应答包，如果Opcode的值为1，那么就是请求包，如果是2，表示应答包。
- 为什么ARP查询要在广播帧中传送，而ARP响应要在一个有着明确目的局域网地址的帧中传送？

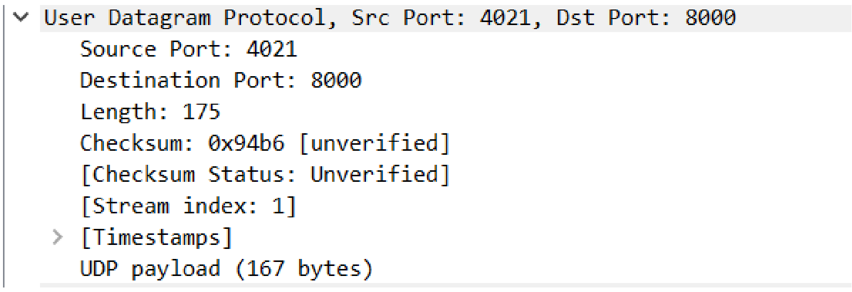
  因为查询主机不知道目的IP地址主机的MAC地址，所以需要查询所有的主机，就要求所有的计算机都能收到查询请求，所以就是以广播帧的形式进行传送。如果目的主机收到了ARP查询请求，而且他就是被查询的，那么他是知道查询主机的IP地址和MAC地址的，所以只需要他返回一个响应即可。

5. 利用Wireshark分析UDP协议

1) 通过QQ给好友发送信息，并捕获，捕获结果如下:



- 可以看出，QQ的消息是基于UDP的。我的主机IP为：172.20.31.84，发送消息的端口号为：4021，目的主机IP地址为：120.53.80.88，接受消息的端口号为8000。
- 数据报头的格式为：



可以看到，UDP数据报报头包含源端口号（2字节）、目的端口号（2字节）、总长度（2字节）、校验和（2字节）

- 为什么发送一个ICQ数据报后，服务器又返回了一个ICQ数据报？和UDP的不可靠数据传输有什么联系？能看出UDP是无连接的吗？

这和UDP的确认机制有关系，UDP的确认机制为：当接收方收到了接收方发送的数据，并且检查校验和，如果无误的话，就会接收数据，并返回一个确认报文。如果有错误，会直接丢失。

这种确认机制，只有当正确收到信息才会返回确认信息，当消息丢失了或者出错了，发送方不会进行重传，而且接收方接受的数据也不是无序的，所以UDP是不可靠数据传输

可以看出UDP是无连接的，我们先来回顾一下TCP连接，TCP传输数据之前，需要进行三次握手建立连接，之后才可以发送数据。但是UDP不需要建立连接，如果需要发送数据，直接发就可以了，说明了它是无连接的。

6. 利用Wireshark进行DNS协议分析
   - DNS查询：

```
Domain Name System (query)
    Transaction ID: 0x6c03
  > Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    > www.bilibili.com: type A, class IN
    [Response In: 13]
```

- DNS响应

```
Domain Name System (response)
    Transaction ID: 0x6c03
  > Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 7
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    > www.bilibili.com: type A, class IN
  v Answers
    > www.bilibili.com: type CNAME, class IN, cname a.w.bilicdn1.com
    > a.w.bilicdn1.com: type A, class IN, addr 111.31.33.19
    > a.w.bilicdn1.com: type A, class IN, addr 111.31.33.21
    > a.w.bilicdn1.com: type A, class IN, addr 111.31.33.20
    > a.w.bilicdn1.com: type A, class IN, addr 111.19.247.143
    > a.w.bilicdn1.com: type A, class IN, addr 111.31.33.18
    > a.w.bilicdn1.com: type A, class IN, addr 111.19.247.142
    [Request In: 11]
    [Time: 0.019442000 seconds]
```

通过Answers我们可以看到解析的结果，这里返回了多个IP，这种情况下，浏览器会根据某种规则选择一个ip进行访问。

实验结果：

具体实现结果以及每个问题在实验过程中解答~~

问题讨论：

1. 在使用WireShark分析IP协议的时候，我捕获的分组头部的校验和一直为0x000，如下图所示：

```
Time to Live: 6
Protocol: ICMP (1)
Header Checksum: 0x0000 [validation disabled]
[Header checksum status: Unverified]
Source Address: 172.20.31.84
Destination Address: 66.39.79.119
```

勾选上下图的选项之后，变为了：

```
Internet Protocol Version 4
[x] Decode IPv4 TOS field as DiffServ field
[ ] Reassemble fragmented IPv4 datagrams
[x] Show IPv4 summary in protocol tree
[x] Validate the IPv4 checksum if possible
[x] Support packet-capture from IP TSO-enabled hardware
```

```
Time to Live: 6
Protocol: ICMP (1)
Header Checksum: 0x0000 incorrect, should be 0x7fe7(may be caused by "IP checksum offload"?)
v [Expert Info (Error/Checksum): Bad checksum [should be 0x7fe7]]
```

一时间搞得我有点蒙，为什么校验和不能够正确的计算出来？

- 经过查阅资料，我了解了原因：出现第一种状况（校验和一直为0x0000）的原因是因为我本地网卡开启了校验和功能，然后WireShark没开启Validate the IPV4 checksum if possible。本地发出去的数据包的校验和，会被填充为0，然后交给网卡硬件完成校验和字段的计算以及值的修改。

  如果我们查看对端发送的数据包，可以看到校验和是正常显示的，有对端网卡硬件完成。

  如果本地网卡开启了校验和功能而且WireShark开启Validate the IPV4 checksum if possible，会出现上述incorrect情况，因为Wireshark抓包是在网卡处理前，所以开启验证的话会认为校验和有问题。

2. 在利用Wireshark对TCP协议进行分析的时候，对含有HTTP POST命令的TCP报文段进行分析，这种情况下发生了分段，按照所学的知识，第一个TCP只发送请求行和请求头，所以长度并不会达到MSS，但是我这里却达到了MSS，还携带了数据。

- 查找相关资料后，我了解到可能有以下情况：

（1） 请求体很小，如果请求体数据很小，他可能会被包含在第一个TCP段中，HTTP允许请求体数据在单个TCP段中传输，如果数据量足够小以适应TCP段的最大大小

（2） HTTP分块传输：HTTP请求可以使用″分块传输″（chunked transfer encoding）来传输数据。在这种情况下，请求体数据可以被分成多个小块，并逐个发送，其中第一个小块可能会包含在第一个TCP段中。

（3） HTTP流水线化：在HTTP/1.1中，允许客户端同时发送多个HTTP请求，而服务器也可以以不同的顺序响应这些请求。这意味着请求体数据可能会与其他请求的头部混合传输，尤其在流水线化的情况下。

心得体会：

在学习、完成本次实验的过程中，我学会了网络协议的分析工具 Wireshark 的使用，对 HTTP、TCP、IP、DNS、UDP、ARP 等协议有了更深入的了解，也学到了一些有关的新的知识。通过 Wireshark 工具，学会了如何使用它来捕获和分析网络数据包，了解了数据包的结构以及如何解释它们，这有助于我更好地理解网络通信的细节，诊断网络问题以及改善网络性能，帮助我深入了解它们的工作原理和交互过程。