# 华东师范大学数据科学与工程学院实验报告

| | | |
|---|---|---|
| 课程名称:计算机网络与编程 | 年级:22级 | 上机实践成绩: |
| 指导教师:张召 | 姓名:郭夏辉 | 学号:10211900416 |
| 上机实践名称:TCP协议分析 | 上机实践日期:2023年5月19日 | 上机实践编号:No.11 |
| 组号:1-416 | 上机实践时间:2023年5月19日 | |

## 一、实验目的

- 了解 TCP 协议的工作原理
- 学习TCP建立连接三次握手的过程
- 学习TCP断开连接四次挥手的过程

## 二、实验任务

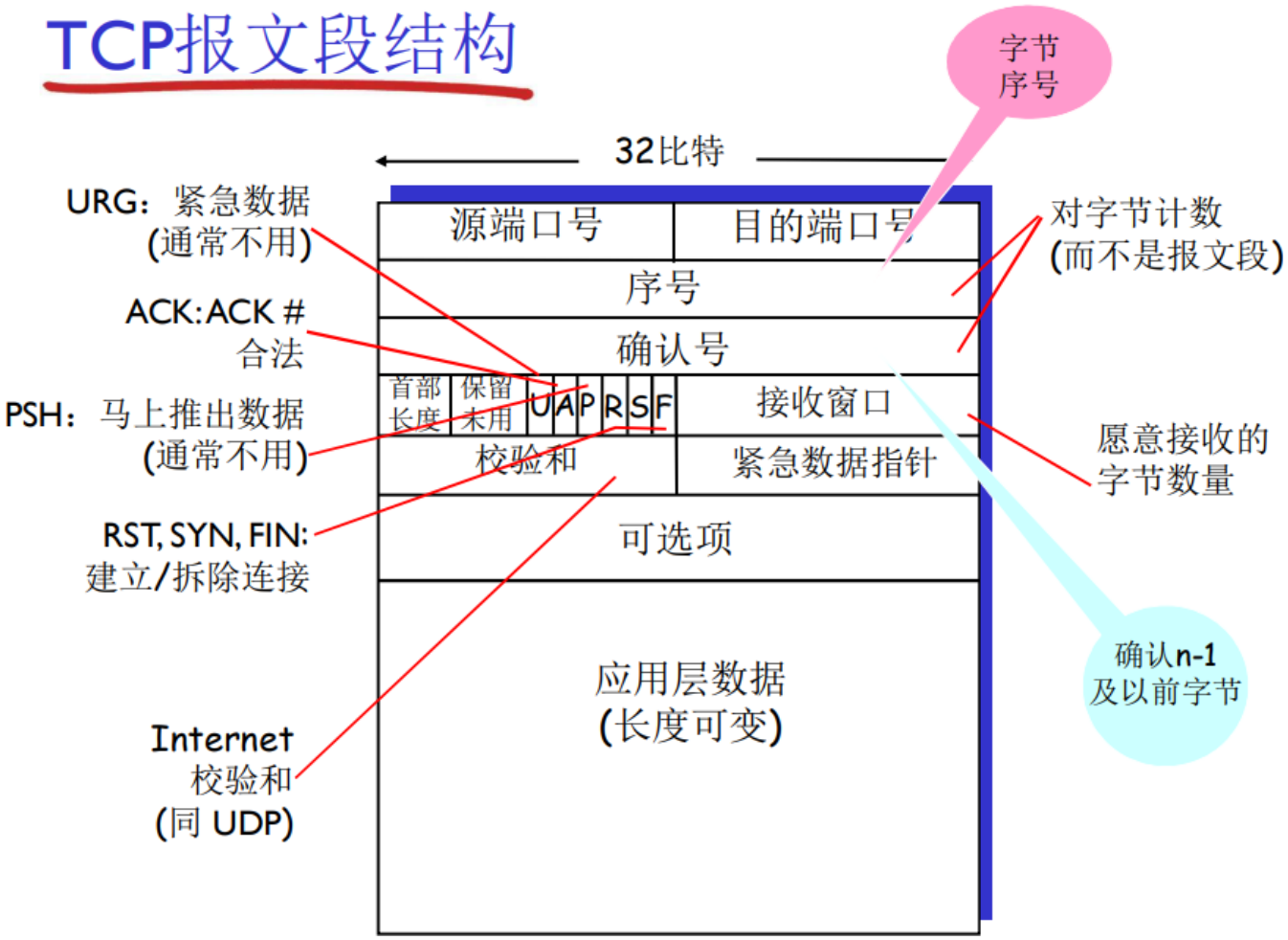
- 使用Wireshark快速了解TCP协议

## 三、实验过程

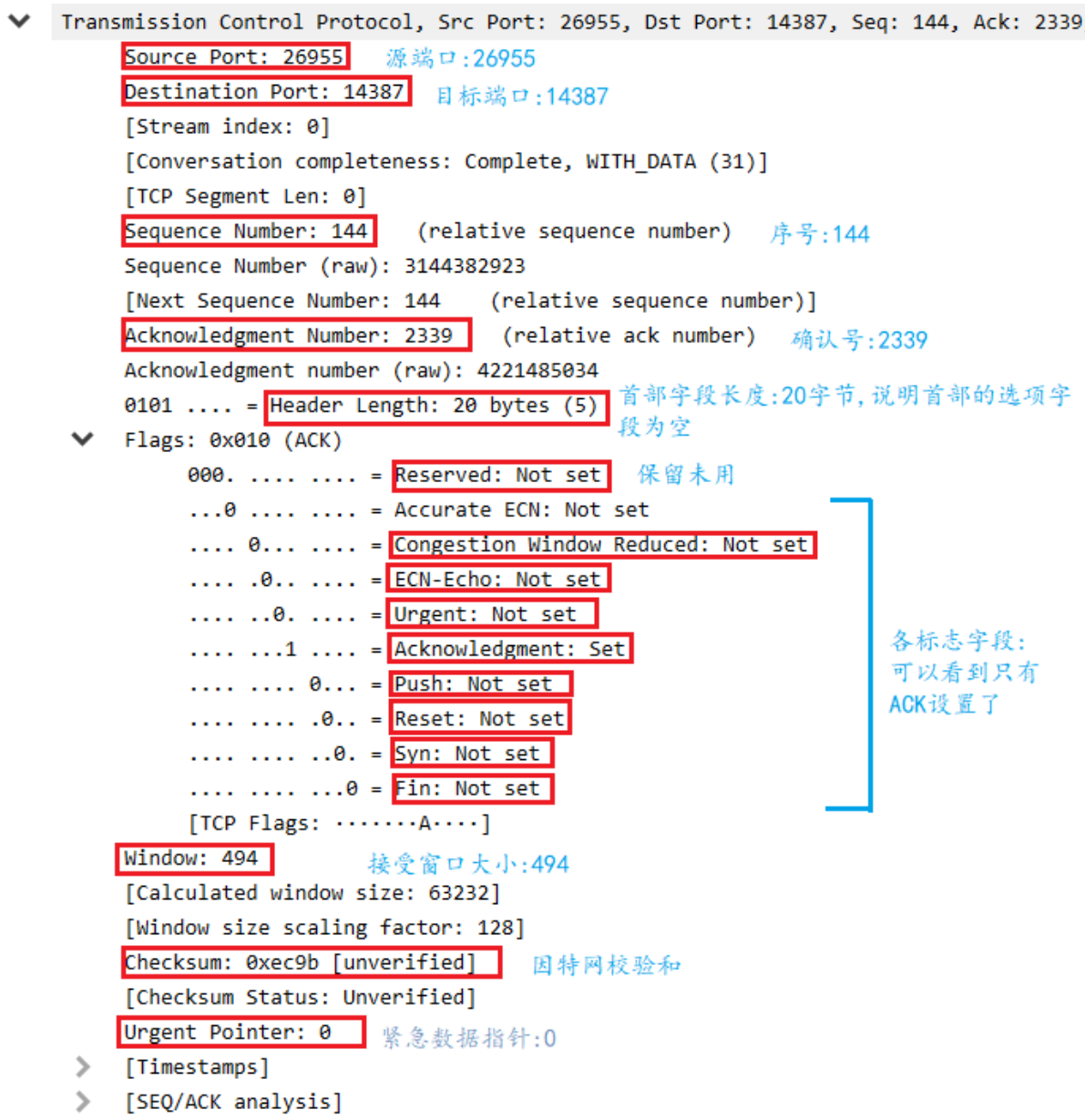
### task1

利用Wireshark抓取一个TCP数据包，查看其具体数据结构和实际的数据（要求根据报文结构正确标识每个部分），请将实验结果附在实验报告中。

首先还是来回顾一下TCP数据包的结构：




可以看到，这个相对于UDP数据包复杂得多，不过还是有很多相近的地方，接下来就让我结合具体的例子来分析一下吧：




这个相对于UDP数据包最大的不一样之处在于少了整个数据包的长度，然后多了很多和连接有关的状态量，比如序号、确认号——毕竟TCP是一种有连接的协议。

然后还有个比较有意思的就是我看到很多"relative"，说明这个序号、确认号是相对的。具体的解释还需要下周我在理论课学习TCP有关原理才能展开，但是我猜想这个是由于TCP的序号是建立在传送的字节流基础上的，而不是建立在传送的报文段的序列之上的，这就涉及到了相对和绝对的问题，确认号应该也与之同理。

而实际的数据是十六进制的，比如源端口:26955(0x694b):

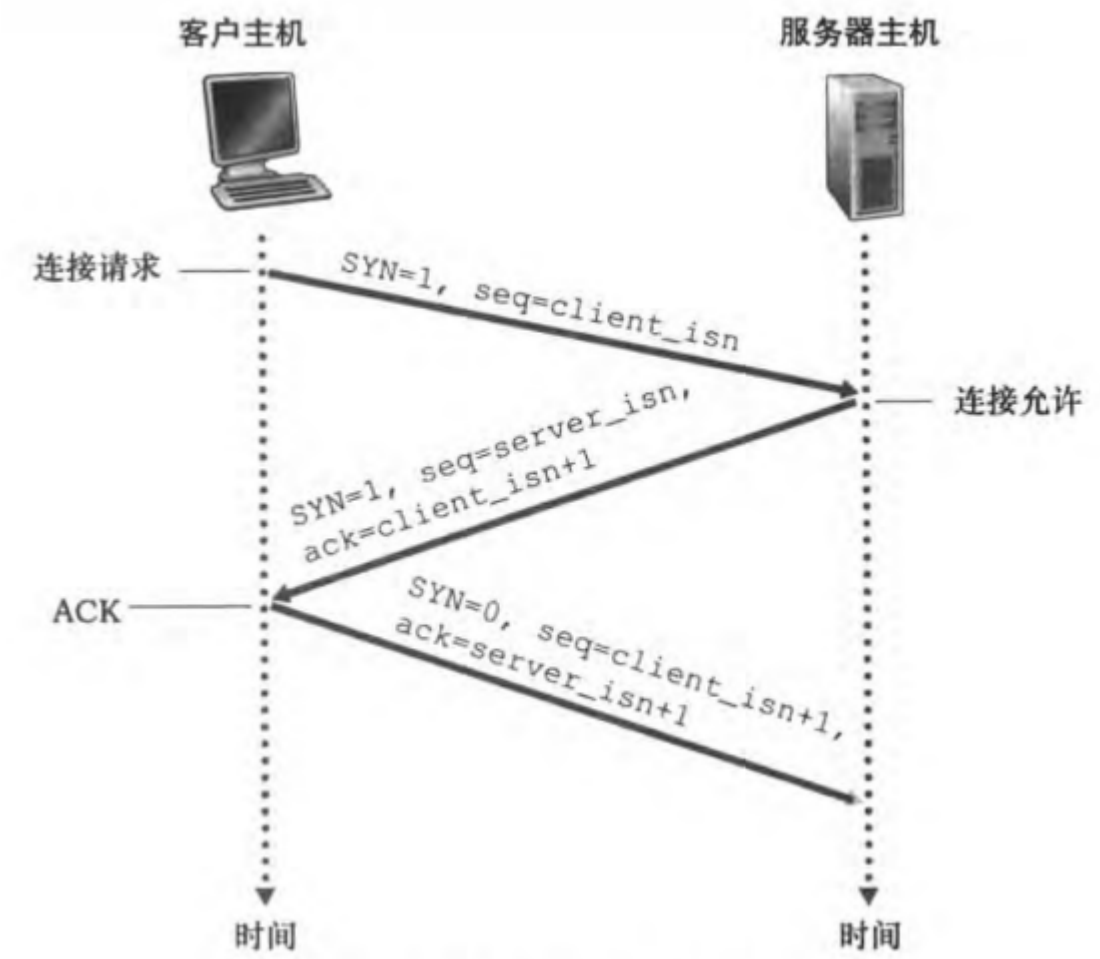
```
38 fc 98 71 57 05 54 c6  ff 7b 38 02 08 00 45 20
00 28 b4 c1 40 00 25 06  71 e3 3b 26 7b 8b ac 1e
cc 3b 69 4b 38 33 bb 6b  79 cb fb 9e bf ea 50 10
01 ee ec 9b 00 00
```

# task2

根据TCP三次握手的交互图和抓到的TCP报文详细分析三次握手过程，请将实验结果附在实验报告中。

TCP建立连接时的三次握手原理:

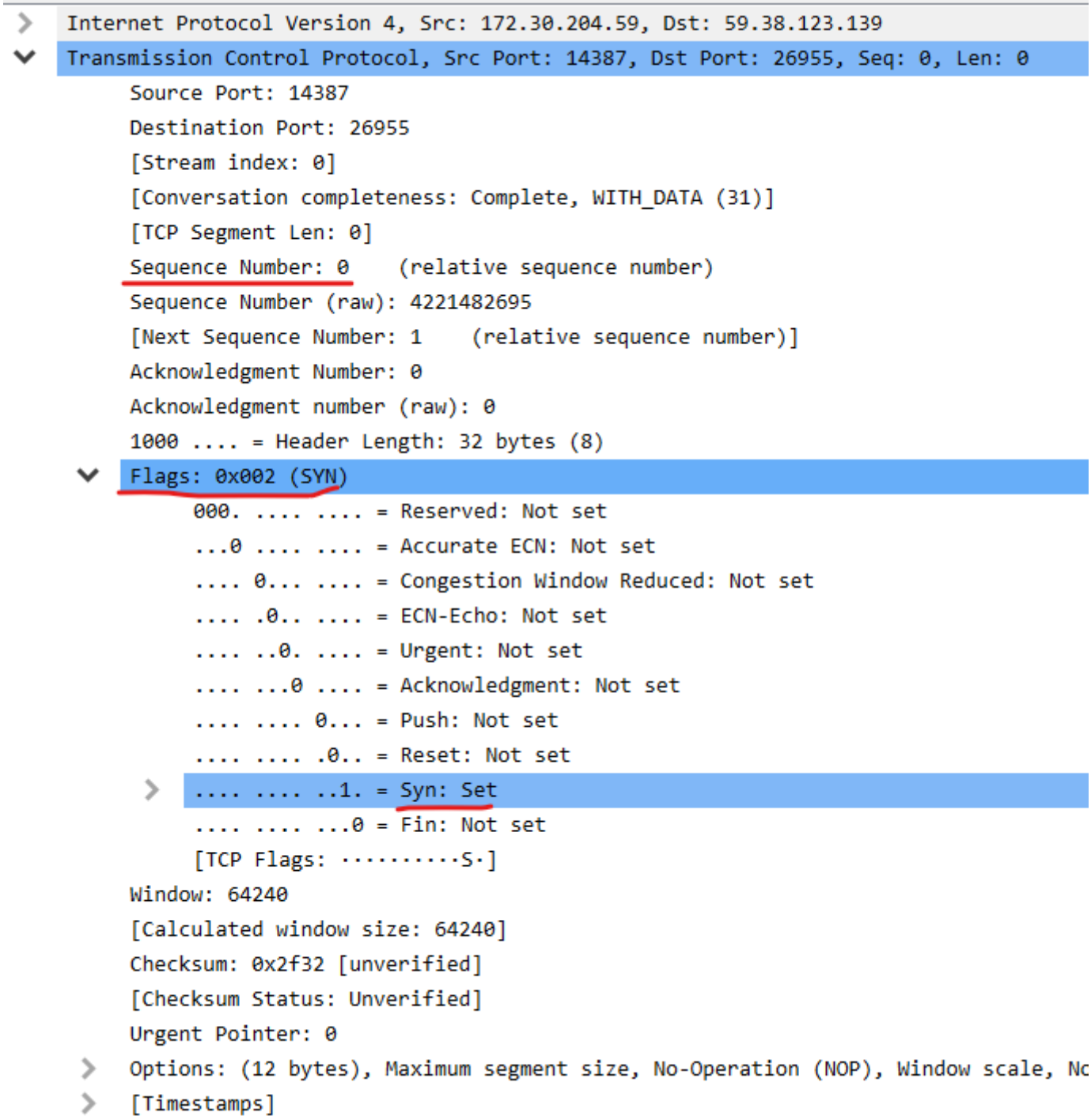
- 第一步：客户端的 TCP 首先向服务器端的 TCP 发送一个特殊的 TCP 报文段。该报文段中不包含应用层数据。但是在报文段的首部（参见图 3-29）中的一个标志位（即 SYN 比特）被置为 1。因此，这个特殊报文段被称为 SYN 报文段。另外，客户会随机地选择一个初始序号（client_isn），并将此编号放置于该起始的 TCP SYN 报文段的序号字段中。该报文段会被封装在一个 IP 数据报中，并发送给服务器。为了避免某些安全性攻击，在适当地随机化选择 client_isn 方面有着不少有趣的研究［CERT 2001-09］。
- 第二步：一旦包含 TCP SYN 报文段的 IP 数据报到达服务器主机（假定它的确到达了!），服务器会从该数据报中提取出 TCP SYN 报文段，为该 TCP 连接分配 TCP 缓存和变量，并向该客户 TCP 发送允许连接的报文段。（我们将在第 8 章看到，在完成三次握手的第三步之前分配这些缓存和变量，使得 TCP 易于受到称为 SYN 洪泛的拒绝服务攻击。）这个允许连接的报文段也不包含应用层数据。但是，在报文段的首部却包含 3 个重要的信息。首先，SYN 比特被置为 1。其次，该 TCP 报文段首部的确认号字段被置为 client_isn + 1。最后，服务器选择自己的初始序号（server_isn），并将其放置到 TCP 报文段首部的序号字段中。这个允许连接的报文段实际上表明了："我收到了你发起建立连接的 SYN 分组，该分组带有初始序号 client_isn。我同意建立该连接。我自己的初始序号是 server_isn。" 该允许连接的报文段被称为 **SYNACK 报文段**（SYNACK segment）。
- 第三步：在收到 SYNACK 报文段后，客户也要给该连接分配缓存和变量。客户主机则向服务器发送另外一个报文段；这最后一个报文段对服务器的允许连接的报文段进行了确认（该客户通过将值 server_isn + 1 放置到 TCP 报文段首部的确认字段中来完成此项工作）。因为连接已经建立了，所以该 SYN 比特被置为 0。该三次握手的第三个阶段可以在报文段负载中携带客户到服务器的数据。




我分析的例子:

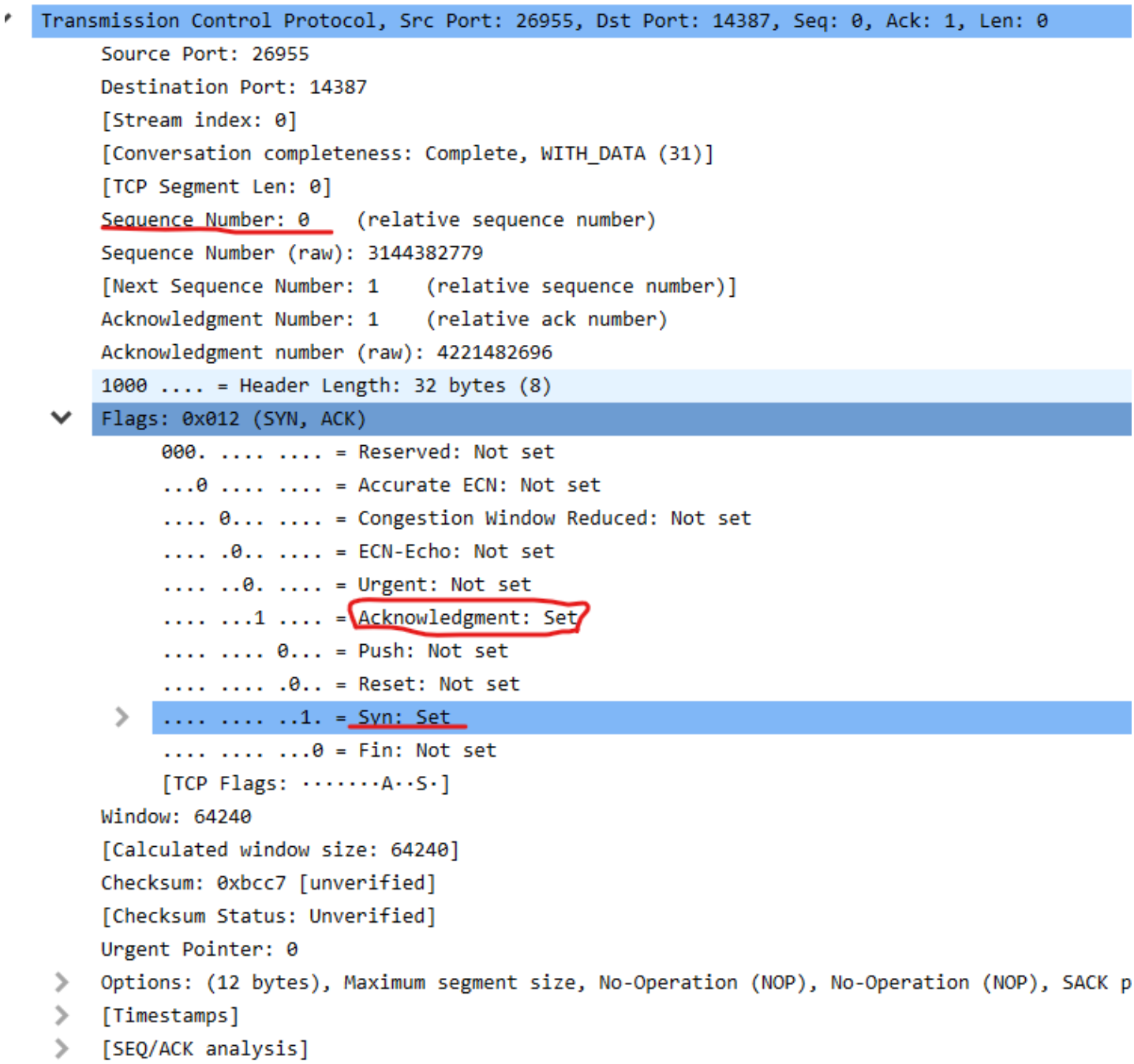
| | | | | | |
|---|---|---|---|---|---|
| 4 | 5.311478 | 172.30.204.59 | 59.38.123.139 | TCP | 66 14387 → 26955 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM |
| 5 | 5.384662 | 59.38.123.139 | 172.30.204.59 | TCP | 66 26955 → 14387 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1380 SACK_PERM WS=128 |
| 6 | 5.384763 | 172.30.204.59 | 59.38.123.139 | TCP | 54 14387 → 26955 [ACK] Seq=1 Ack=1 Win=131072 Len=0 |
| 7 | 5.385187 | 172.30.204.59 | 59.38.123.139 | TLSv1.2 | 613 Client Hello |

第一次握手:




这时主要的工作是将SYN标志位设置为1，然后我们也可以看到这时序号seq为0，说明最初是从序号为0的包开始发送的；这时还没有收到服务器的响应，因此ack还为0。

第二次握手:




这时终于收到了服务器端的肯定了，所以ACK被设置为了1.

最开始的确认号被设置为了1(0+1=1)，由于TCP是累计确认的，这个意思是说1之前的数据包我已经收到并确认了，请你传之后的包。

有个值得注意的地方是这里的序号seq还是0，这说明服务器端的初始序号也是0，最初也是从序号为0的包开始发送的。

第三次握手:

```
Transmission Control Protocol, Src Port: 14387, Dst Port: 26955, Seq: 1, Ack: 1, Len: 0
    Source Port: 14387
    Destination Port: 26955
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 4221482696
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3144382780
    0101 .... = Header Length: 20 bytes (5)
    Flags: 0x010 (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
        .... .... ..0. = Syn: Not set
        .... .... ...0 = Fin: Not set
        [TCP Flags: ·····A····]
    Window: 512
    [Calculated window size: 131072]
    [Window size scaling factor: 256]
    Checksum: 0x2f26 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    [Timestamps]
    [SEQ/ACK analysis]
```

这次握手是用户端发给服务器端的。有几个变化就是SYN重新被置为了0,由于是对服务器端的允许连接的报文进行确认，ACK还是被设置为1.

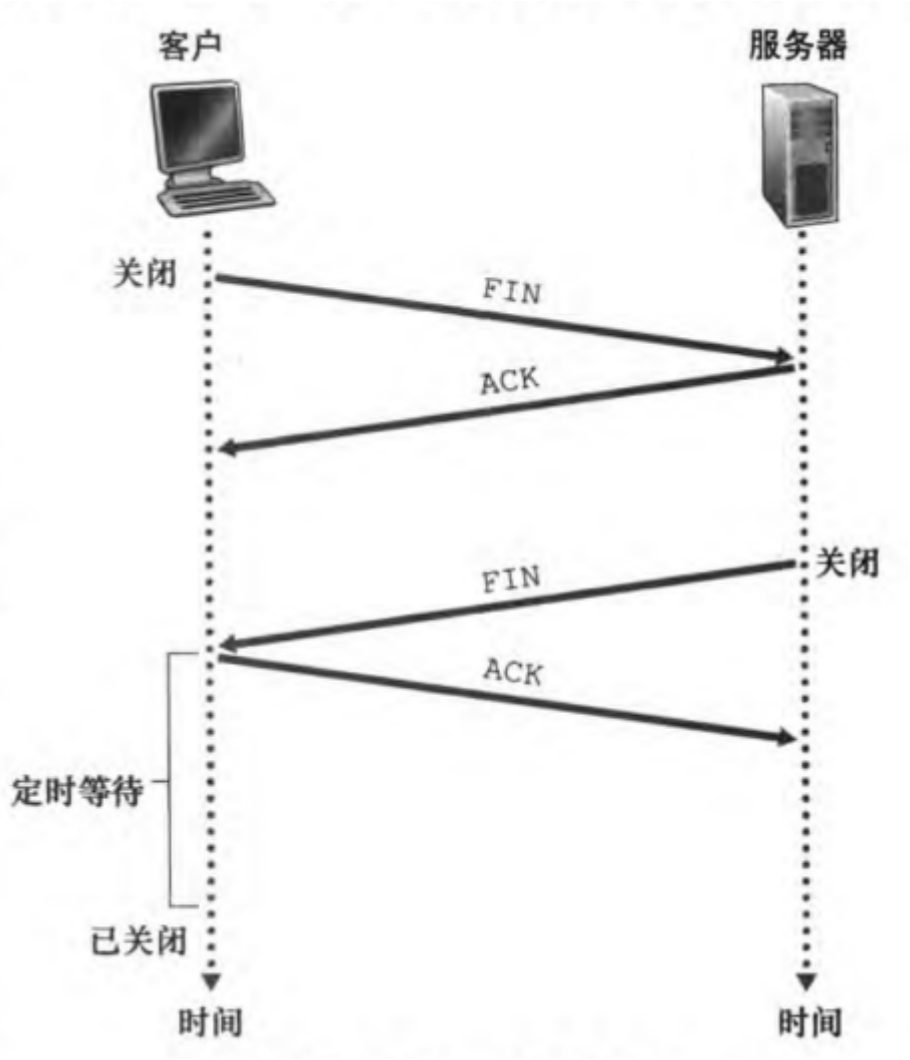
序号seq已经被设置为了1,表达的是向服务器端传送的第二个包。确认号也已经被设置为了1，意思是说1之前的数据包我已经收到并确认了，请你传之后的包。

一条TCP连接就这样建立起来了！

## task3

根据TCP四次挥手的交互图和抓到的TCP报文详细分析四次挥手过程，请将实验结果附在实验报告中。

在关闭一条TCP连接时，会发生四次挥手:

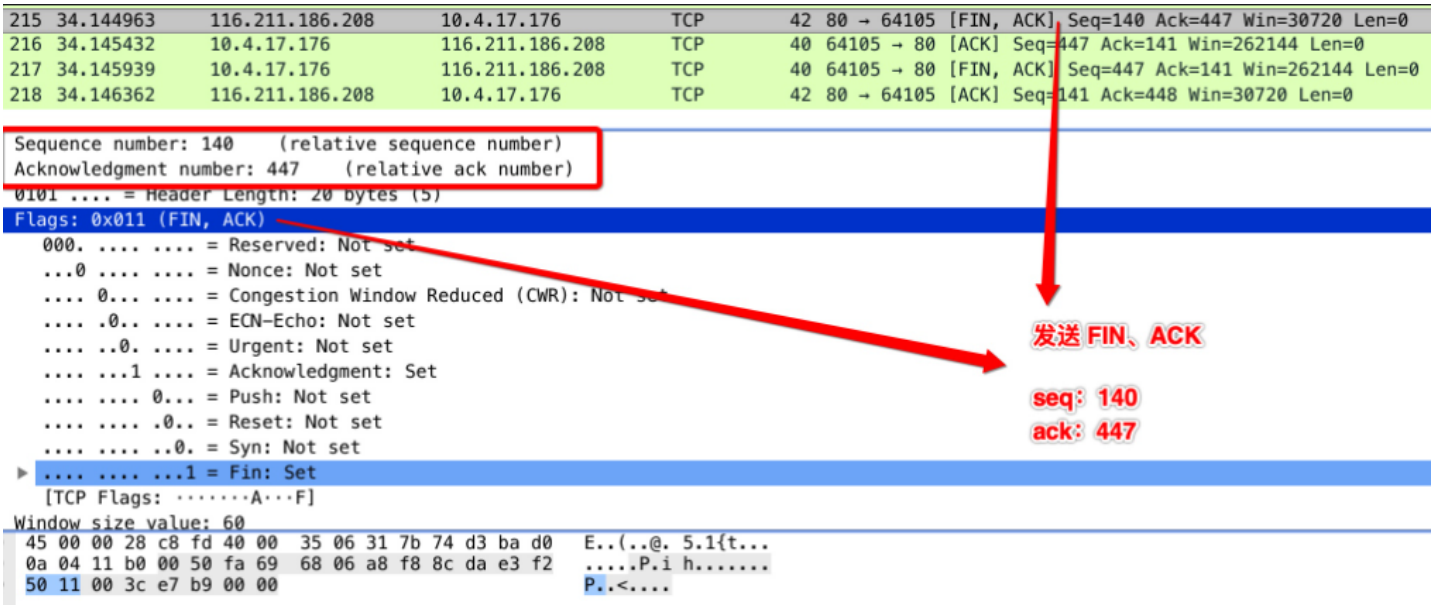



做这个实验尽量不要用大网站，因为使用大网站的话可能有多个服务端建立起了多个TCP连接，最后分析TCP连接关闭很容易非常混乱!

我结合的例子:

```
116.211.186.208     10.4.17.176         TCP       42 80 → 64105 [FIN, ACK] Seq=140 Ack=447 Win=30720 Len=0
10.4.17.176         116.211.186.208     TCP       40 64105 → 80 [ACK] Seq=447 Ack=141 Win=262144 Len=0
10.4.17.176         116.211.186.208     TCP       40 64105 → 80 [FIN, ACK] Seq=447 Ack=141 Win=262144 Len=0
116.211.186.208     10.4.17.176         TCP       42 80 → 64105 [ACK] Seq=141 Ack=448 Win=30720 Len=0
```

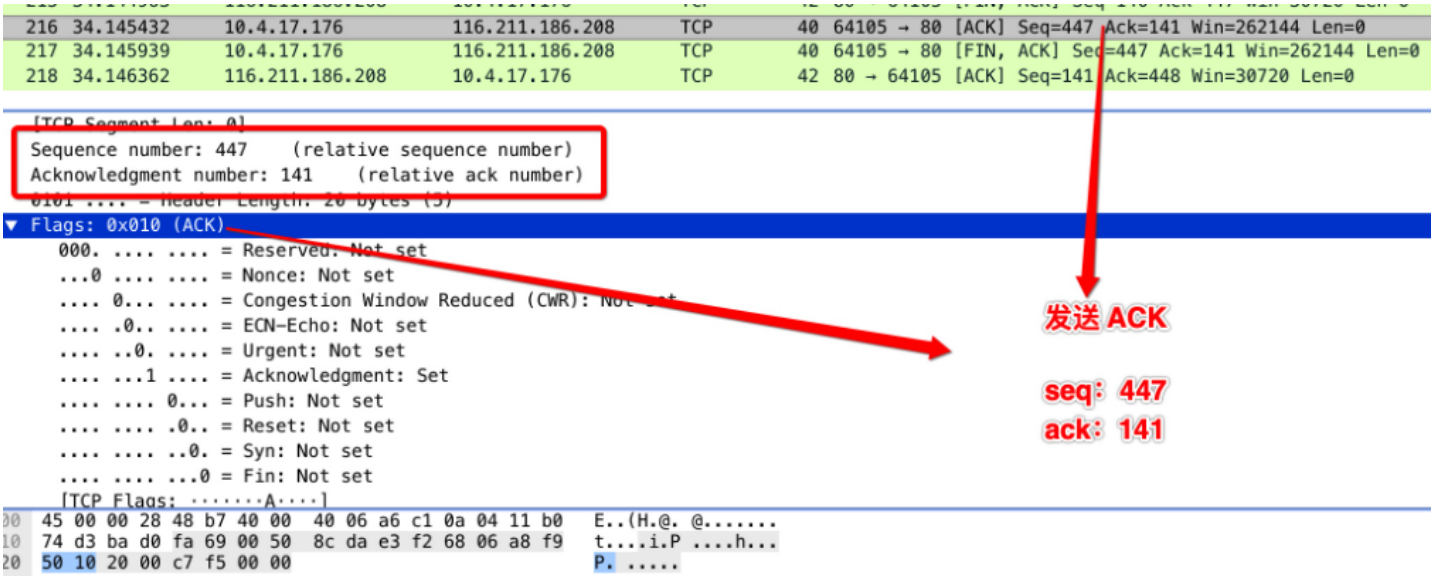
第一次挥手:




主动停止端发送 `FIN` 和 `ACK` 报文

服务端发送一个 `[FIN+ACK]` 报文，表示自己没有数据要发送了，想断开连接，并进入 `fin_wait_1` 状态（不能再发送数据到客户端，但能够发送控制信息 `ACK` 到客户端）。

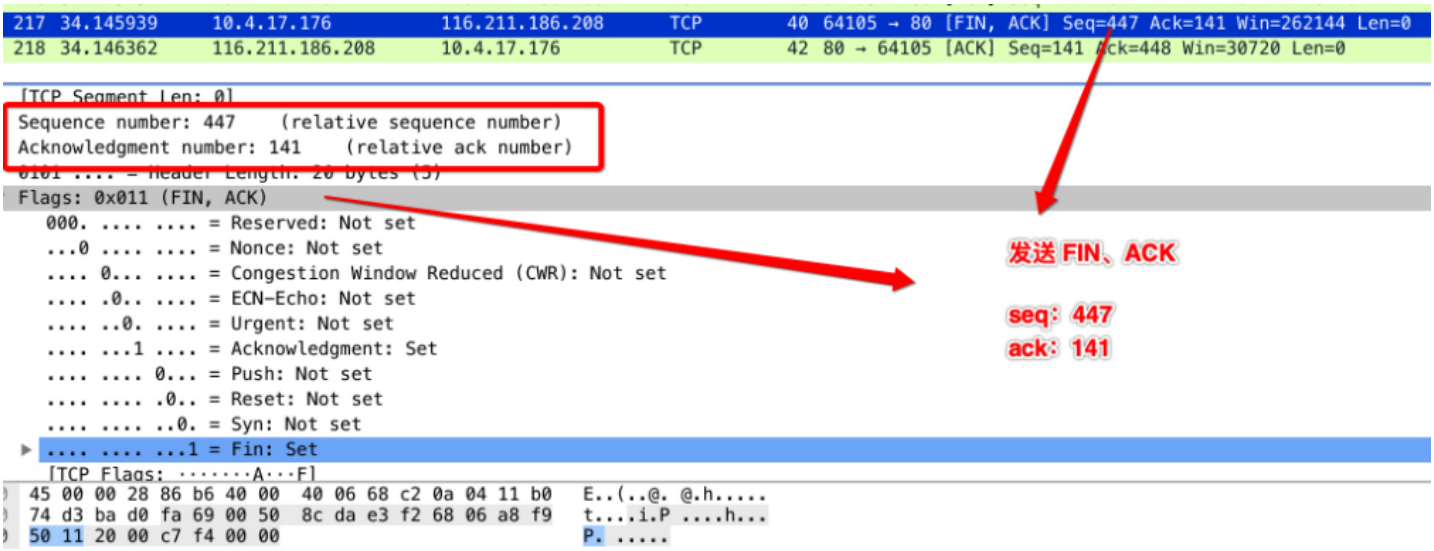
第二次挥手:




被动停止端发送 `ACK` 报文给主动停止端

客户端收到 `[FIN]` 报文后，客户端知道不会再有数据从服务端传来，发送 `ACK` 进行确认，客户端进入 `close_wait` 状态。此时服务端收到了客户端对 `FIN` 的 `ACK` 后，进入 `fin_wait2` 状态。

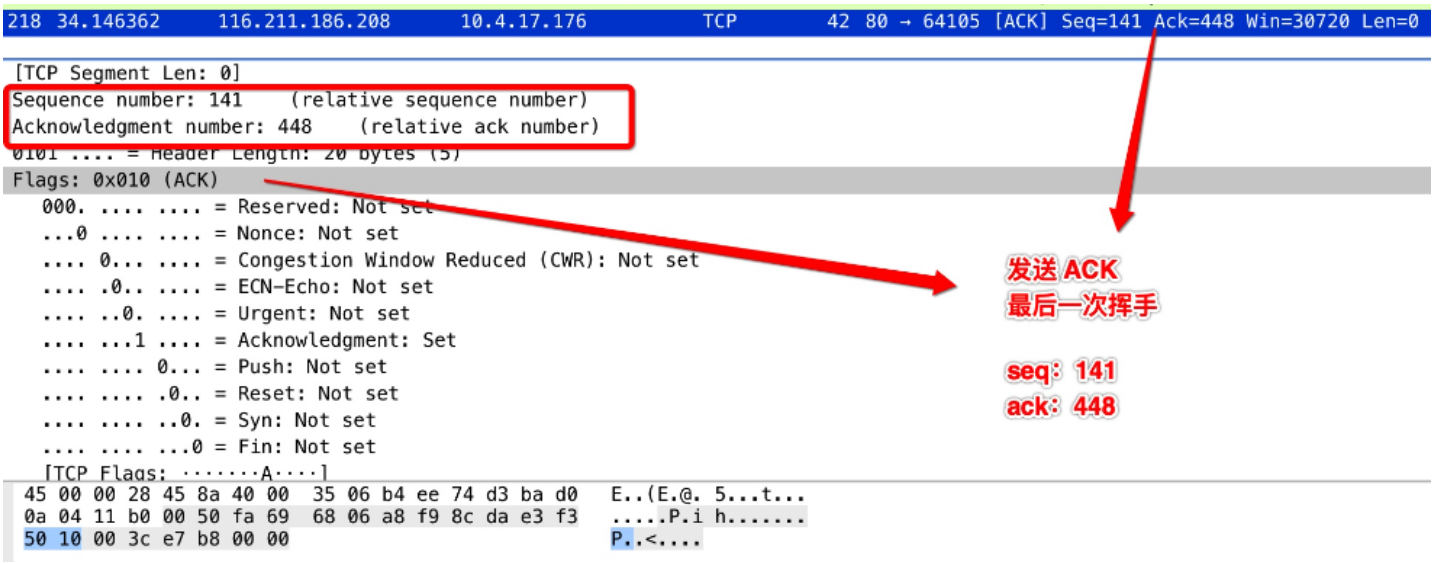
第三次挥手:




被动停止端发送 `FIN` 和 `ACK` 报文给主动停止端

客户端发送 `[FIN ACK]` 报文给对方，表示自己没有数据要发送了，客户端进入 `last_ack` 状态。服务端收到了客户端的 `FIN` 信令后，进入 `time_wait` 状态，并发送 `ACK` 确认消息。

第四次挥手:




主动停止端发送 `ACK` 报文给被动停止端

服务端在 `time_wait` 状态下，等待一段时间，没有数据到来的，就认为对面已经收到了自己发送的 `ACK` 并正确关闭了进入 `close` 状态，自己也断开了到客户端的 `TCP` 连接，释放所有资源。当客户端收到服务端的`ACK` 回应后，会进入 `close` 状态，并关闭本端的会话接口，释放相应资源。

为了更明确化相关的序号seq和确认号ack之变化，我做了个小表格:

| 时间 | 序号**SEQ** | 确认号**ACK** |
| --- | --- | --- |
| 1 | 140 | 447 |

| 时间 | 序号**SEQ** | 确认号**ACK** |
| --- | --- | --- |
| 2 | 447 | 141 |
| 3 | 447 | 141 |
| 4 | 141 | 448 |

其实这个seq和ack变化的原理和task2中建立连接我已经详细说了，此处就不展开说了。

一条TCP连接终于关闭了！

注意,断开连接端可以是 `Client` 端，也可以是 `Server` 端，我上面的例子首先发起 `close` 的一方是 `Server` 端。

## 四、总结

这是一次难度不算大，但是十分有趣的实验。通过Wireshark抓包，TCP的建立和关闭连接的过程被直观地展现在我的面前。经过深入细致的实践，我对TCP/UDP这两个传输层协议有了更深刻的认识，希望我能在未来娴熟有效地利用好这两者，在计算机网络的世界中畅游！