Lab2 实验报告

姓名：吴静

专业：信息安全

学号：2113285

一、实验要求

1.搭建 Web 服务器(自由选择系统)，并制作简单的 Web 页面，包含简单文本信息(至少包含专业、学号、姓名)、自己的 LOGO、自我介绍的音频信息。页面不要太复杂，包含要求的基本信息即可。

2.通过浏览器获取自己编写的 Web 页面，使用 Wireshark 捕获浏览器与 Web 服务器的交互过程，并进行简单的分析说明.

3.使用 HTTP，不要使用 HTTPS。

4.提交实验报告

二、实验步骤

1.搭建 web 服务器

使用平台：xampp

编写简单的 html 代码 start.html，代码如下：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>计网第二次作业</title>
</head>
<body>
    <h1>计网第二次作业</h1>
<p>姓名：吴静</p>
```

```
<p>学号：2113285</p>
<p>专业：信息安全</p>
<img src="./wanderer.jpg" alt="小圆帽" width="304" height="304">

<p>introduction</p>
<audio controls>
    <source src="./introduction.mp3" type="audio/mp3">
</audio>

</body>
</html>
```

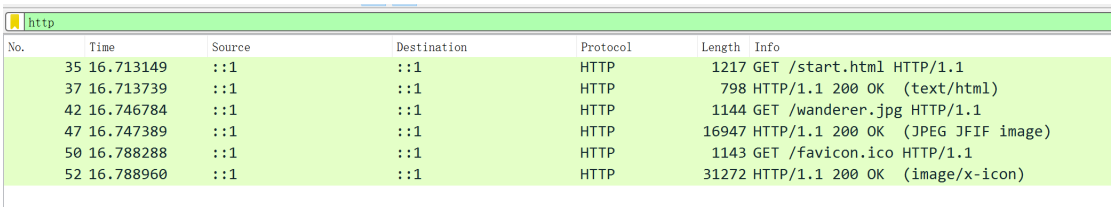
打开 xampp，然后在 web 服务器里面输入“localhost:8081/start.html”即可出现页面，截图如下：

2. 捕获交互过程

首先打开 wireshark 软件，由于客户端和服务器都是主机，所以选择 loopback 进行数据包的捕获。

然后在网页中打开对应的网址，在 wireshark 中设置过滤规则为“http”，表示查看捕获到的 http 数据包，接下来进行 wireshark 捕获到的数据包的分析。

3. 数据包分析——http 相关数据包

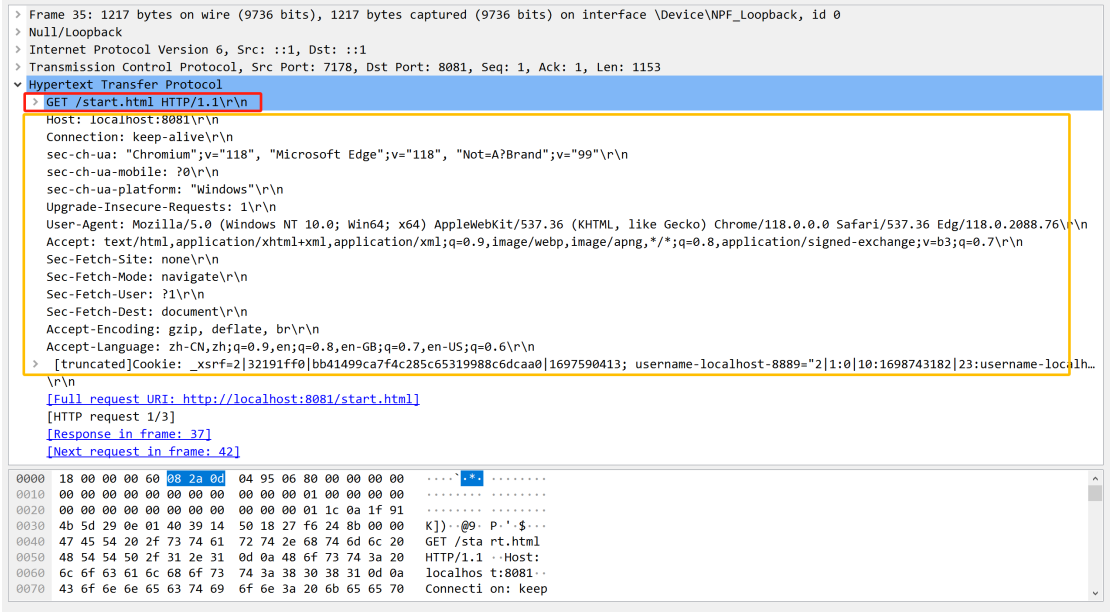


http

No.	Time	Source	Destination	Protocol	Length	Info
35	16.713149	::1	::1	HTTP	1217	GET /start.html HTTP/1.1
37	16.713739	::1	::1	HTTP	798	HTTP/1.1 200 OK (text/html)
42	16.746784	::1	::1	HTTP	1144	GET /wanderer.jpg HTTP/1.1
47	16.747389	::1	::1	HTTP	16947	HTTP/1.1 200 OK (JPEG JFIF image)
50	16.788288	::1	::1	HTTP	1143	GET /favicon.ico HTTP/1.1
52	16.788960	::1	::1	HTTP	31272	HTTP/1.1 200 OK (image/x-icon)

这是一些 HTTP 请求和响应的示例，以第一二句为例，首先点进“GET /start.html HTTP/1.1”查看，查询有关该请求报文的内容：

（1）HTTP 请求报文



```
Frame 35: 1217 bytes on wire (9736 bits), 1217 bytes captured (9736 bits) on interface \Device\NPF_Loopback, id 0
Null/Loopback
Internet Protocol Version 6, Src: ::1, Dst: ::1
Transmission Control Protocol, Src Port: 7178, Dst Port: 8081, Seq: 1, Ack: 1, Len: 1153
Hypertext Transfer Protocol
  GET /start.html HTTP/1.1\r\n
  Host: localhost:8081\r\n
  Connection: keep-alive\r\n
  sec-ch-ua: "Chromium";v="118", "Microsoft Edge";v="118", "Not=A?Brand";v="99"\r\n
  sec-ch-ua-mobile: ?0\r\n
  sec-ch-ua-platform: "Windows"\r\n
  Upgrade-Insecure-Requests: 1\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.0.0 Safari/537.36 Edg/118.0.2088.76\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7\r\n
  Sec-Fetch-Site: none\r\n
  Sec-Fetch-Mode: navigate\r\n
  Sec-Fetch-User: ?1\r\n
  Sec-Fetch-Dest: document\r\n
  Accept-Encoding: gzip, deflate, br\r\n
  Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6\r\n
  [truncated]Cookie: _xsrf=2|32191ff0|bb41499ca7f4c285c65319988c6dcaa0|1697590413; username-localhost-8889="2|1:0|10:1698743182|23:username-localh…
  \r\n
  [Full request URI: http://localhost:8081/start.html]
  [HTTP request 1/3]
  [Response in frame: 37]
  [Next request in frame: 42]
```

```
0000  18 00 00 00 60 08 2a 0d  04 95 06 80 00 00 00 00   ····`·*· ········
0010  00 00 00 00 00 00 00 00  00 00 00 01 00 00 00 00   ········ ········
0020  00 00 00 00 00 00 00 00  00 00 00 01 1c 0a 1f 91   ········ ········
0030  4b 5d 29 0e 01 40 39 14  50 18 27 f6 24 8b 00 00   K])··@9· P·'·$···
0040  47 45 54 20 2f 73 74 61  72 74 2e 68 74 6d 6c 20   GET /sta rt.html
0050  48 54 54 50 2f 31 2e 31  0d 0a 48 6f 73 74 3a 20   HTTP/1.1 ··Host:
0060  6c 6f 63 61 6c 68 6f 73  74 3a 38 30 38 31 0d 0a   localhos t:8081··
0070  43 6f 6e 6e 65 63 74 69  6f 6e 3a 20 6b 65 65 70   Connecti on: keep
```

http 请求报文分成三个部分：请求行（request line），请求头部（head），空行和请求数据四个部分。

请求行即图中的红色部分，由请求方法字段，URL 字段和 HTTP 协议版本字段 3 个字段组成，由空格分隔。客户端发起了一个 GET 请求，请求的 URL 是/start.html，而且使用的是 HTTP/1.1 协议版本。

接下来的部分是**请求头部**，即图中的黄色部分。请求头部由关键字/值对组成，每行一对（由”\r\n”结尾）关键字和值用英文冒号”:”分隔，他通知了服务器有关客户端的请求的信息。举例如下：

host 表示请求的主机名；

connection 表示连接方式，keep-alive 指示连接应保持活动；

Sec-Ch-UA 字段通常包括浏览器的名称和版本以及其他特定的客户端信息；

Accept 字段用于指定客户端可以接受的响应内容类型，包括多个内容类型，按优先级排序；

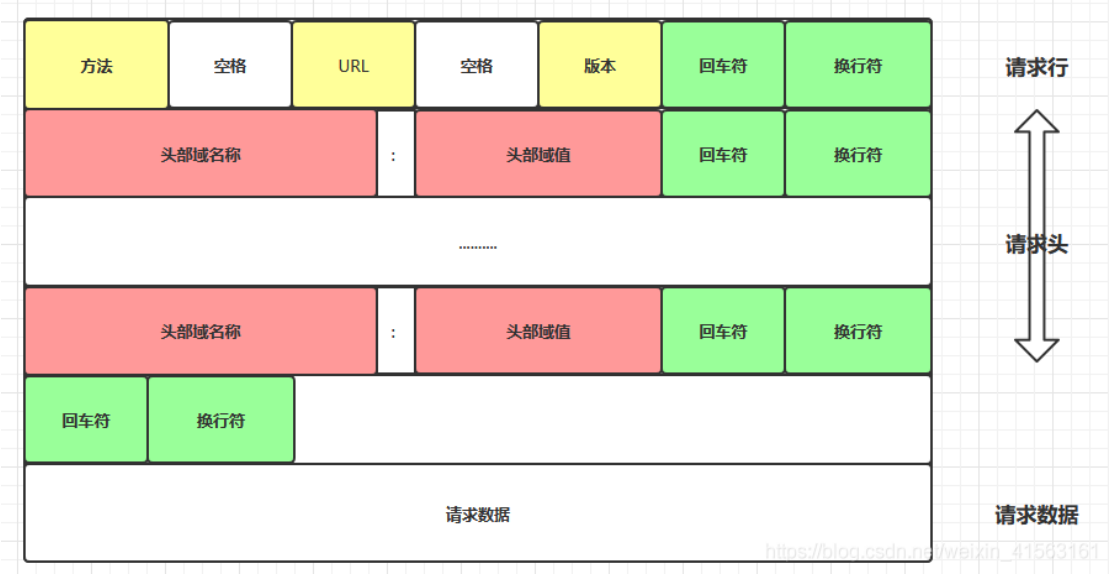
User-Agent：产生请求的浏览器类型；

Cookie：存储于客户端扩展字段，向同一域名的服务端发送属于该域的 cookie；

可以看到，请求头部后面跟着一个小**空行**表示以下不再有请求头部发内容。

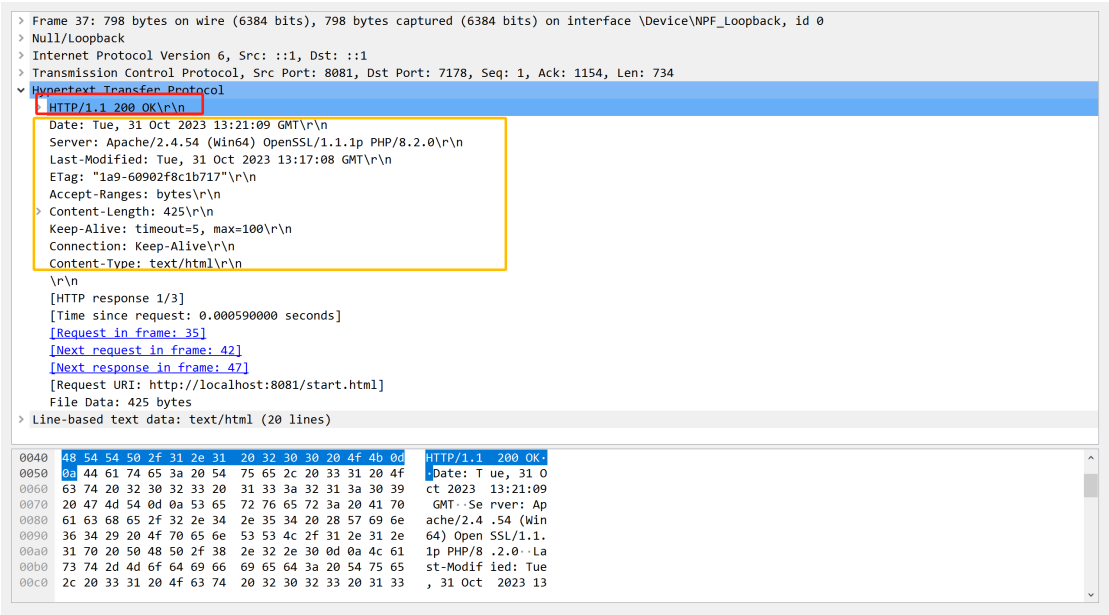
后面没有**请求体**的部分，因为此时的请求方法是 GET，没有请求体部分。

结构如下：



（2）HTTP 响应报文

例如图中的“HTTP/1.1 200 OK（text/html）”字段，同样点进去查看。



响应报文同样由四个部分组成：响应行，响应头，空行和响应数据。

响应行，即图中的红色部分由三个部分组成：服务器 HTTP 协议版本，响应状态码，状态码的文本描述。经过查阅得知，响应状态码有五种取值：“1xx”表示请求已接受，“2xx”

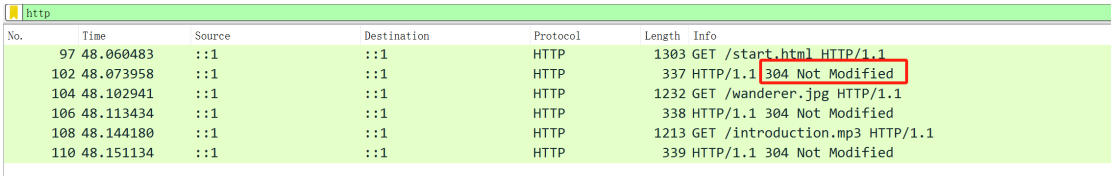
表示请求被成功接收；“3xx”表示发生了重定向，“4xx”表示客户端错误，“5xx”表示服务器端错误；常见的有：

响应状态码：“200”，文本描述：“OK”，表示客户端请求成功。

响应状态码：“304”，文本描述：“Not Modified”，表示资源未被修改，客户端可以使用缓存的版本。

响应状态码：“404”，文本描述：“Not Found”，表示资源不存在。

下图中即为未清除缓存时出现 304 响应的情况。



http

No.	Time	Source	Destination	Protocol	Length	Info
97	48.060483	::1	::1	HTTP	1303	GET /start.html HTTP/1.1
102	48.073958	::1	::1	HTTP	337	HTTP/1.1 304 Not Modified
104	48.102941	::1	::1	HTTP	1232	GET /wanderer.jpg HTTP/1.1
106	48.113434	::1	::1	HTTP	338	HTTP/1.1 304 Not Modified
108	48.144180	::1	::1	HTTP	1213	GET /introduction.mp3 HTTP/1.1
110	48.151134	::1	::1	HTTP	339	HTTP/1.1 304 Not Modified

图中的黄色部分为**响应头**。也是由多个属性构成，举例如下：

Date 表示响应时间；

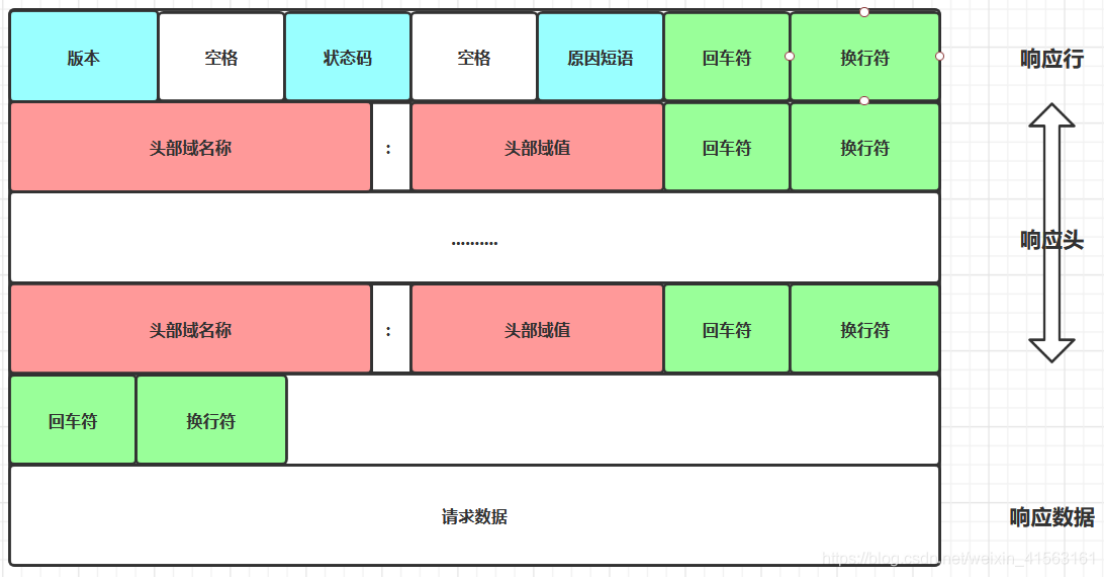
Server 表示服务器应用程序软件的名称和版本；

Accept-Ranges 表示对此资源来说，服务器可接受的范围类型；

Content-Length 表示实体的长度，指示了客户端应该期望接收多少字节的响应数据；

Content-Tyep 表示实体的媒体类型；

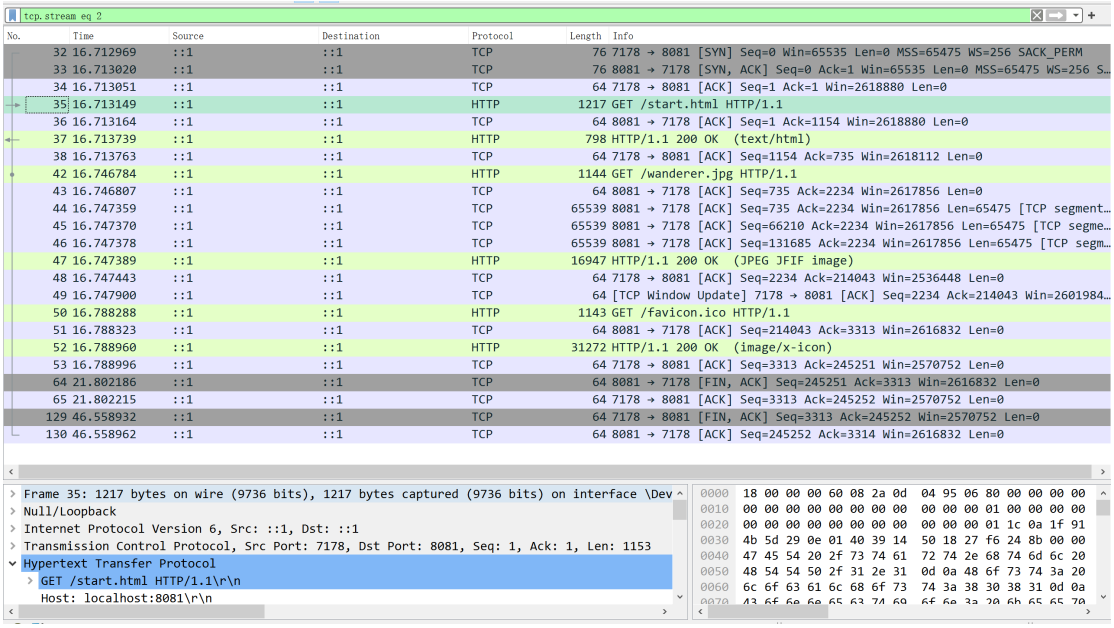
格式如下：



这里注意”\r\n”后的部分不再是响应报文中的部分，二是 wireshark 列出来帮助解释和分析捕获的数据包，比如[HTTP response 1/3]表示总共有 3 个 HTTP 响应，这是第一个响应；[Request in frame: 35]：这表明 HTTP 请求在捕获的数据包的第 35 帧中。

4. 传输过程分析——传输控制协议 TCP

首先随机选择一个 HTTP 响应/请求报文，右键选中“追踪流”，选中追踪 TCP 流，得到如下界面：

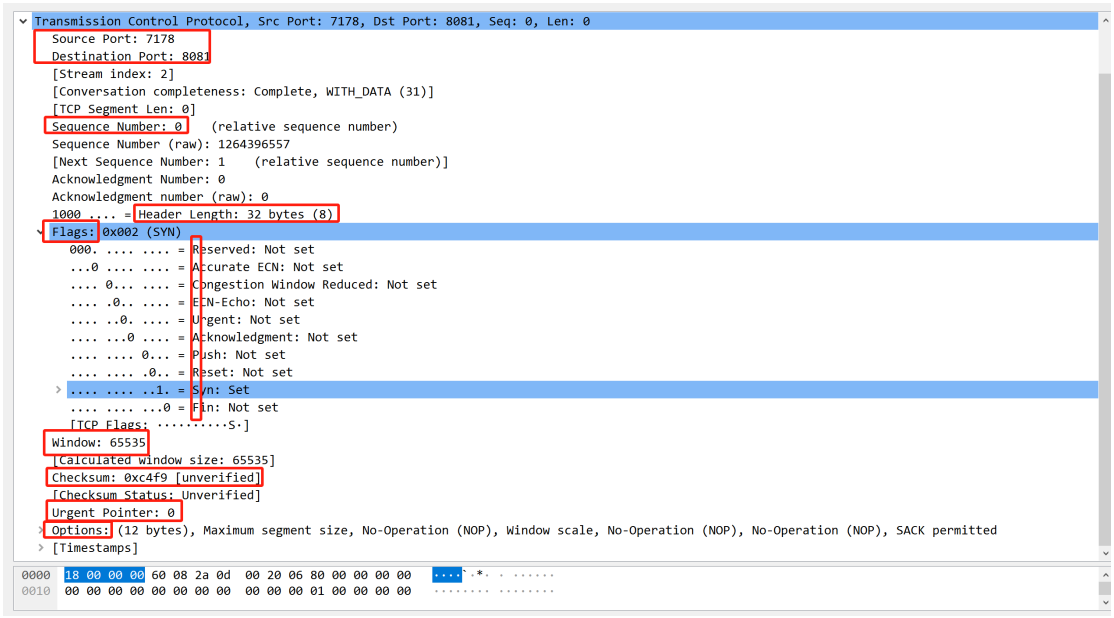


我们知道，TCP 是面向连接的服务，在传送数据之前必须先建立连接，数据传送完后要释放连接，而在建立连接的时候要经过三次握手的过程，释放连接的时候要经过四次挥手，这一个部分，我讲重点分析这三次握手和四次挥手的过程。

（1）TCP 报文格式

首先先来看 TCP 报文格式：

同样随机选择一个 TCP 报文，点进去查看：



Source Port：源端口

Destination Port：目的端口

Sequence Number：发送序号，TCP 字节流中的每个字节都按顺序进行编号，这里从 0 开始是因为这是使用的相对的编号。

Header Length：头长度

Flags：其中有 6 个 bit 的保留位，以及“U”“A”“P”“R”“S”“F”这六个保留位。

①Urgent 表示紧急指针字段是否有效，如果置位，则表示他告诉系统此报文中有紧急数据，需要尽快传送；

②ACK 为确认位，仅当 ACK 置位时确认号字段 ack 才有效，注意在连接建立后，TCP 规定会把所有的报文段的 ACK 都置 1，由于采用累计确认的方式，ack 通常等于上一个数据包的序列号的值加 1；

③RST 表示复位，表示 TCP 连接中出现严重差错需要释放连接在重新建立运输连接；

④SYM 为同步标志；

⑤最后的 FIN 表示终止，用来释放一个连接。

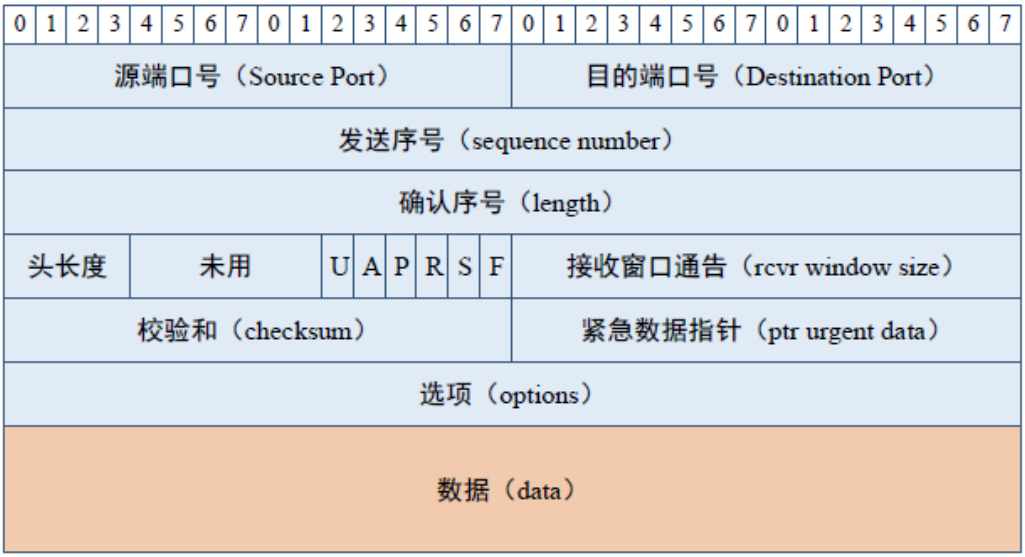
Window：接收窗口通告，即告诉接收方，发送本报文需要多大的空间来接受。

Checksum：校验和，用于检验首部和数据两个部分。

Urgent Pointer：紧急数据指针，指出本报文段中的紧急数据的字节数。

Options：定义一些其他可选的参数（长度可变，此时为 12 字节）

总结来说，结构如下：



接下来，我们将根据这个结构，联系三次握手和四次挥手的具体过程进行分析。

（2）三次握手

三次握手如图所示。

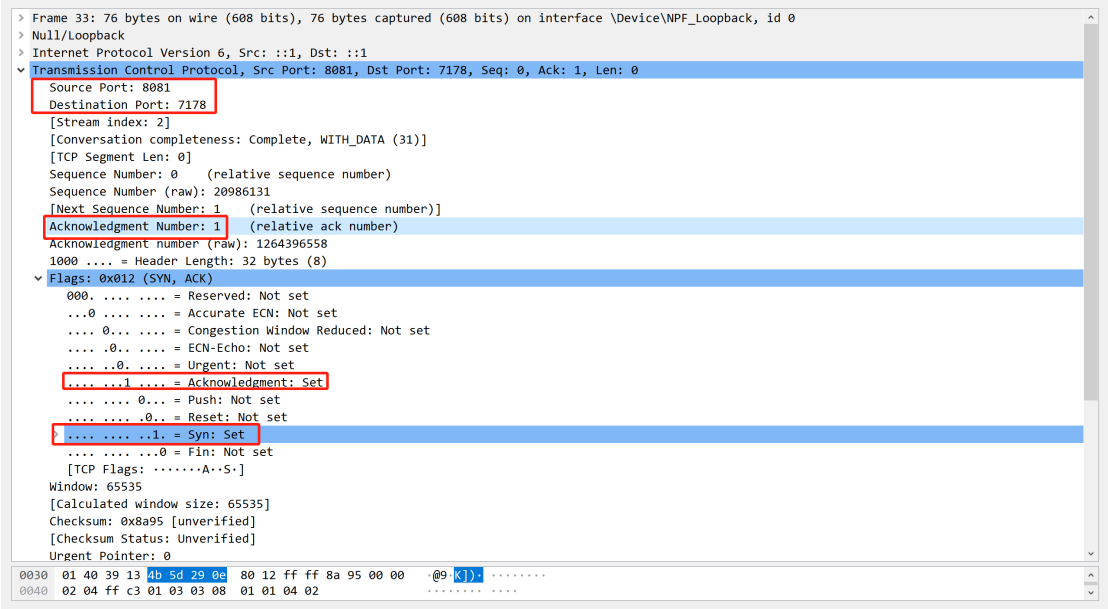
①服务器端进入 LISTEN 监听状态

②客户进程向服务器发出连接请求报文，即该报文的 SYN 位置 1，同时选择一个初始序列号 seq=x。发送完 SYN 后，该客户进程进入 SYN_SENT 状态（同步已发送状态），等待服务器端确认（该报文即为上图所示报文）

③服务器收到请求报文（SYN 位置 1 的报文）后，如果同意连接，则发出确认报文，确认报文中应将 ACK 置 1，将 SYN 置 1，同时确认号 Acknowledgement Number 应该为 x+1（可以看到图中为 1），即收到的连接请求报文的 seq 的值加上 1；

同时，为自己初始化一个序列号 seq=y（此时图中显示为 1）；

服务器进入 SYN-RCVD 状态（同步收到状态）；



④客户进程收到确认后，还要向服务器给出确认。发送确认报文，ACK 位置 1，Acknowledgement Number 为 y+1（此时图中为 1），自己的序列号为 x+1（此时图中为 1）。建立 TCP 连接，客户端进入 ESTABLISHED（已建立连接状态）。

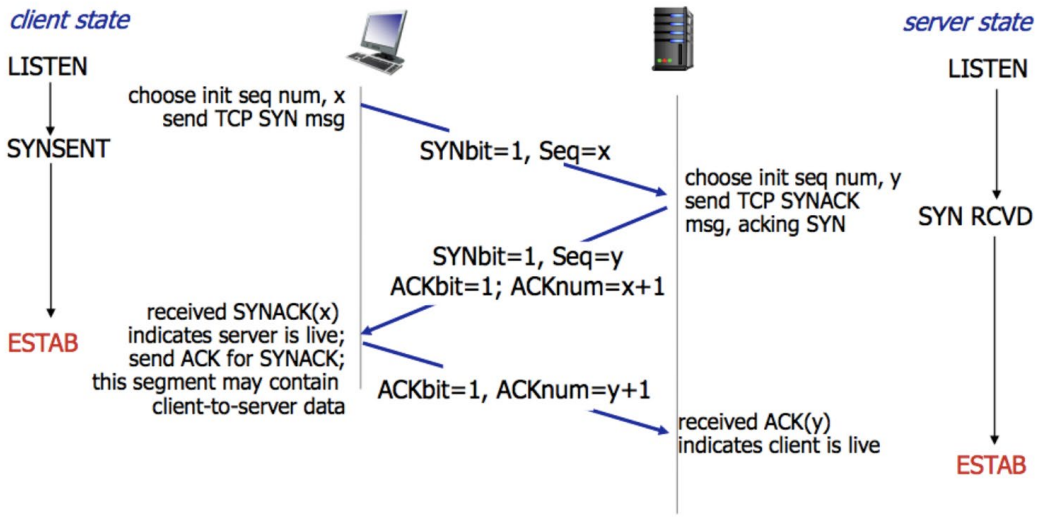
```
> Frame 34: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface \Device\NPF_Loopback, id 0
> Null/Loopback
> Internet Protocol Version 6, Src: ::1, Dst: ::1
v Transmission Control Protocol, Src Port: 7178, Dst Port: 8081, Seq: 1, Ack: 1, Len: 0
    Source Port: 7178
    Destination Port: 8081
    [Stream index: 2]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 1264396558
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 20986132
    0101 .... = Header Length: 20 bytes (5)
  v 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: 10230
    [Calculated window size: 2618880]
    [Window size scaling factor: 256]
    Checksum: 0x9d82 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
```

⑤服务器端收到客户端的确认后也进入 ESTABLISHED 状态，连接建立。

总结：

在建立过程中总共出现了三个类型的报文：客户进程发送的请求报文，确认报文和服务器发送的确认报文。客户进程先发送了一个请求报文，其中 SYN=1，且选择了一个初始序列号 seq=x；然后服务器端发送确认报文，其中 ACK=1，SYN=1，（因为 ACK=1 了所以要关注 ack 即确认序号 Acknowledgement Number），ack=x+1，seq=y；然后客户端再次发送一个确认报文，其中 ACK=1，ack=y+1，seq=x+1。

图解如下：



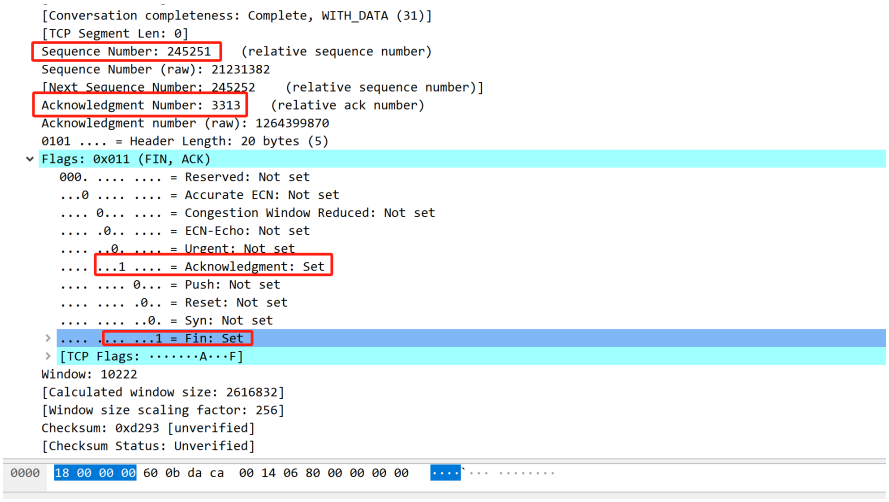
（3）四次挥手

四次挥手如图所示。（注意常见的是客户端先发送连接释放报文，但是本例中的服务器先发送连接释放报文，源端口和目的端口不一样，但是大致的过程是一样的，我的分析以常

见情况为准，而不是以服务器发起连接释放报文为准）

64	21.802186	::1	::1	TCP	64 8081 → 7178 [FIN, ACK] Seq=245251 Ack=3313 Win=2616832 Len=0
65	21.802215	::1	::1	TCP	64 7178 → 8081 [ACK] Seq=3313 Ack=245252 Win=2570752 Len=0
129	46.558932	::1	::1	TCP	64 7178 → 8081 [FIN, ACK] Seq=3313 Ack=245252 Win=2570752 Len=0
130	46.558962	::1	::1	TCP	64 8081 → 7178 [ACK] Seq=245252 Ack=3314 Win=2616832 Len=0

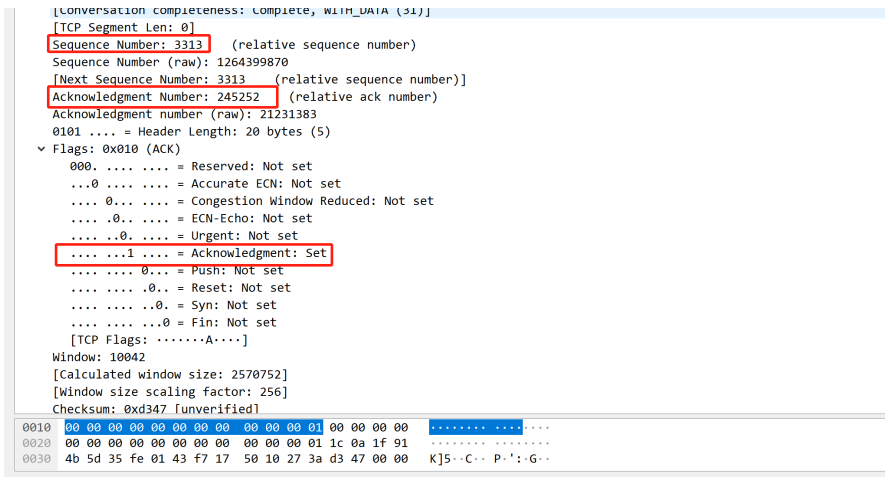
① 客户端进程发出连接释放报文，并且停止发送数据。该连接释放报文的 FIN 位置 1，（此时 ACK 位也可能置 1，是因为这是对上一个包的确认，与四次挥手并没有关系）序列号 seq=u（前面已经传送过来的数据的最后一个字节的序号加 1，此时为 245251），同时，客户端进入 FIN-WAIT-1（终止等待 1）状态，对应的报文如下：



② 服务器端收到连接释放报文，发出确认报文（ACK=1,ack=u+1，此时为 245252），序列号 seq=v（此时为 3313），服务器端进入 CLOSE-WAIT 状态（关闭等待状态），表明自己接收到了客户端关闭连接的请求，但还没有准备好关闭连接。

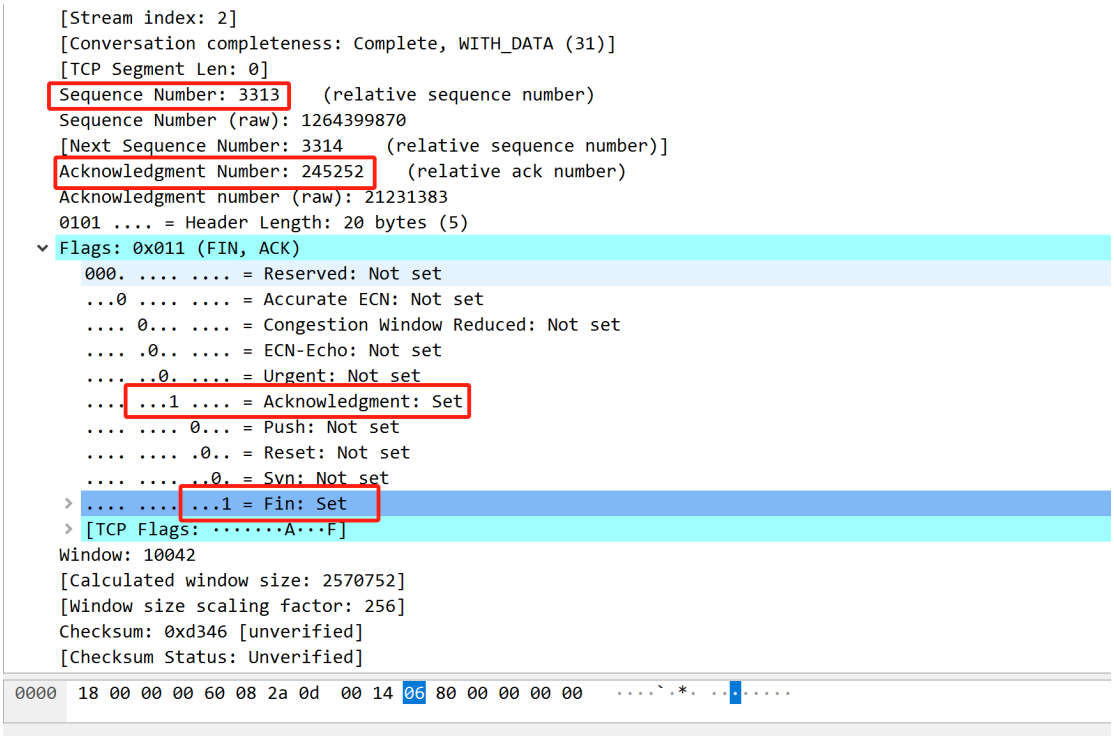
此时处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端还要继续接受。

对应报文如下：



③ 客户端收到确认包后，进入 FIN-WAIT-2 状态（终止等待 2 状态），等待服务器关闭连接。

④ 服务器端准备好关闭连接后（数据全都发送完毕），向客户端发送结束连接请求，发送一个连接释放报文，该报文 FIN=1,ACK=1，ack=u+1（此时为 245252），此时的序列号 seq=w（此时为 3313）；接下来，服务器进入了 LAST-ACK 状态（最后确认状态），等待客户端的确认。对应报文如下：



⑤ 客户端收到服务器的连接释放报文后，发出确认报文，该报文 ACK=1，ack=w+1（此时为 3314），自己的序列号 seq=u+1（此时为 3314），客户端进图 TIME-WAIT 状态（时间等待状态），但是这个状态不等于 TCP 连接的释放，还需要等待 2*MSL（最长报文寿命）的时间后，才进入 CLOSED 状态。对应报文如下：

```
[Stream index: 2]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 245252    (relative sequence number)
Sequence Number (raw): 21231383
[Next Sequence Number: 245252    (relative sequence number)]
Acknowledgment Number: 3314    (relative ack number)
Acknowledgment number (raw): 1264399871
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: 10222
[Calculated window size: 2616832]
[Window size scaling factor: 256]
```

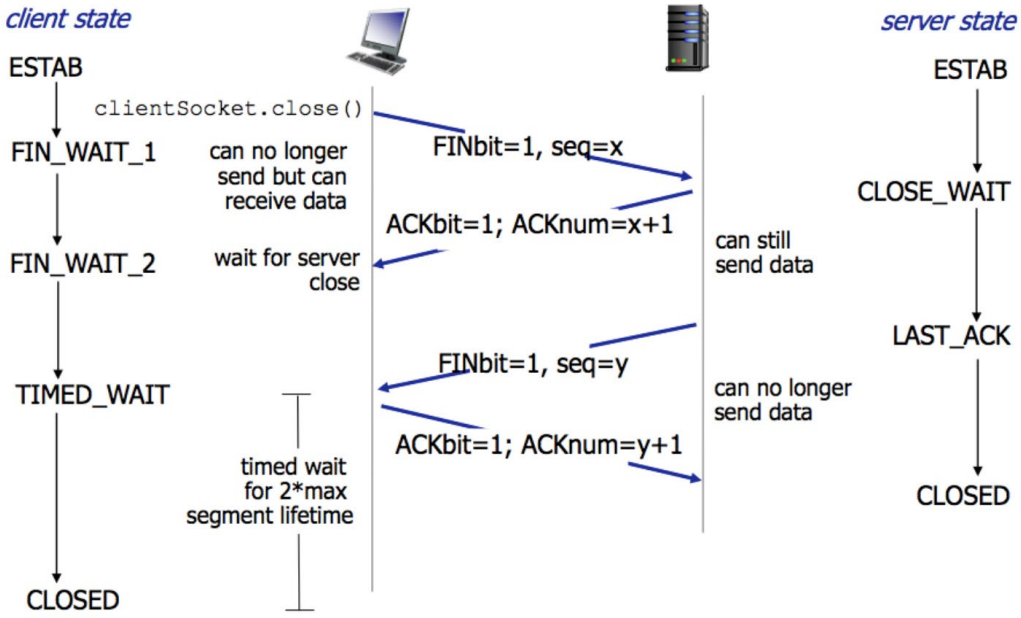
```
0010  00 00 00 00 00 00 00 00  00 00 00 01 00 00 00 00   ........ ........
0020  00 00 00 00 00 00 00 00  00 00 00 01 1f 91 1c 0a   ........ ........
0030  01 43 f7 17 4b 5d 35 ff  50 10 27 ee d2 92 00 00   ·C··K]5· P·'·····
```

⑥服务器收到客户端发出的确认后，立即进入 CLOSED 状态。

总结：

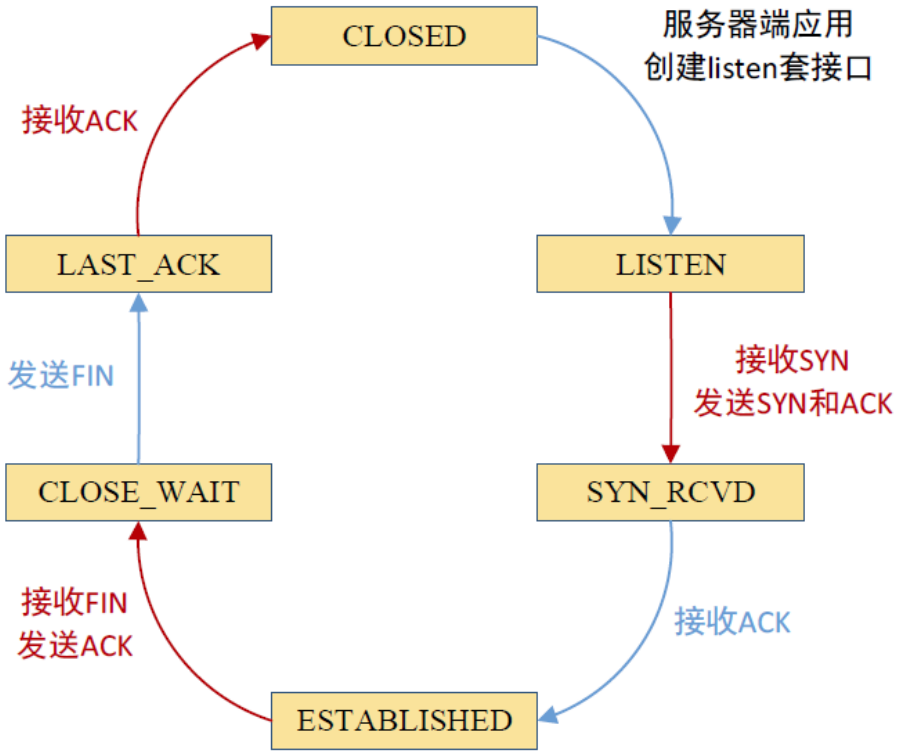
四次挥手可以由客户端发起也可以由服务器端发起，假设 A 端先挥手。A 端首先发送连接释放报文，其中 FIN=1，seq=u，表示这一段没有什么数据要继续发送了；B 端接收到之后，发送确认报文表示成功收到，ACK=1，ack=u+1；当 B 端的数据也发送完成后，发送结束连接请求，其中 FIN=1，ACK=1，ack=u+1；然后 A 端接收到之后，发出确认报文，ACK=1，ack=w+1，seq=u+1，同时进入 TIME-WAIT 状态，等待 2*MSL 时间后进入 CLOSED 状态，而 B 端收到确认报文后立刻进入 CLOSED 状态。

图解如下：

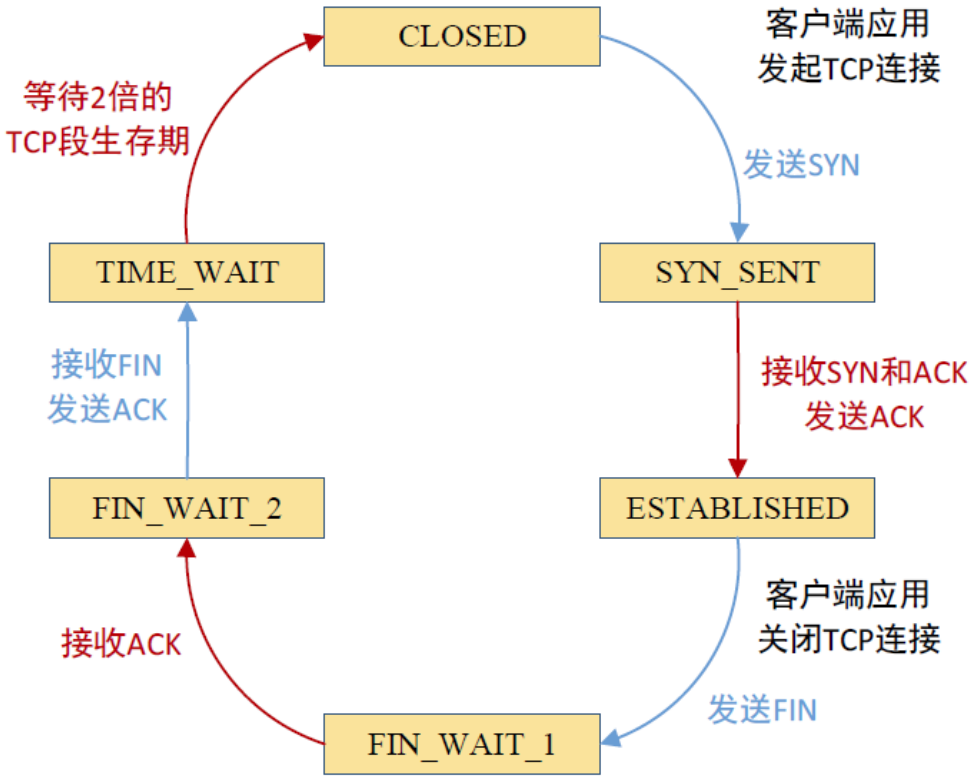


（4）服务器和客户端建立连接和释放连接的过程中的流程图。

服务器端：



客户端：



三、实验思考

1. 建立连接时为什么 TCP 客户端最后还要发送一次确认

防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

2. 如果已经建立了连接，但是客户端突然出现故障了会发生什么

服务器端每隔一段时间会发送一个探测报文，该探测报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。

3. 为什么建立连接时三次握手，关闭连接确是四次挥手

建立连接的时候，服务器在 LISTEN 状态下，收到建立连接请求的 SYN 报文后，把 ACK 和 SYN 放在一个报文里发送给客户端。

而关闭连接时，服务器收到对方的 FIN 报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送 FIN 报文给对方来表示同意现在关闭连接，因此，己方 ACK 和 FIN

一般都会分开发送，从而导致多了一次。

4.有时候只出现了三次挥手是为什么

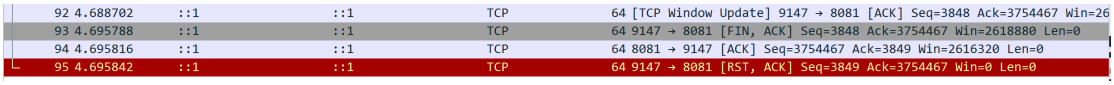
实际上是将第二次挥手和第三次挥手合并了，即服务器端已经没有数据可以发送了，当客户端发送关闭连接请求的时候，服务器端将 ACK（确认包）和 FIN（关闭连接）一起发送，于是表象上看起来只有三次挥手。

5.为什么客户端最后还要等待 2MSL

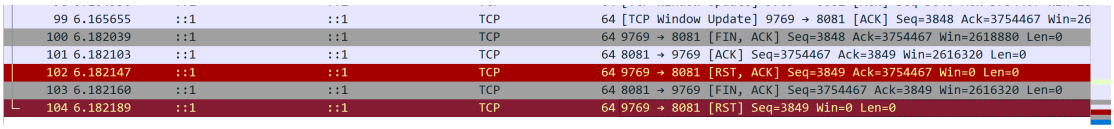
首先，保证客户端发送的最后一个 ACK 报文能够到达服务器。当客户端已经发送数据包但是没有接收到确认时，服务器端就会重新发送一次，此时就有可能在 2MSL 时间内到达客户端，接着给出回应报文，并目重启 2MSL 计时器。

其次，防止类似与“三次握手”中提到了的"已经失效的连接请求报文段”出现在本连接中。

6.客户端发起 FIN 关闭连接请求后，发送了一个 RST 数据包如下图所示。



或者下图所示。



首先，产生 RST 的三个情况是：

（1）目的地为某端口的 SYN 到达，然而在该端口上并没有正在监听的服务器；

（2）TCP 想取消一个已有连接；

（3）在一个已关闭的 TCP 连接上收到数据

图中所示为第三种情况，客户端先向服务器发送了一个 FIN 包，然后服务器确认接受，发送了 ACK，此时只进行了两次挥手，服务器并没有发送 FIN；后服务器继续向客户端发送数据，但是客户端已经关闭，由于此时连接状态异常，所以客户端不会向服务器发送 ACK 字段，而是发送一个 RST 报文请求将处于异常状态的连接复位。

7. http1.0 版本和 1.1 版本有什么区别

http1.0 版本是短链接，每个 HTTP 请求/响应都需要建立一个新的 TCP 连接；

http1.1 版本是长连接，允许多个请求和响应共享同一连接。

8. 一张图片为什么会被分段传输（TCP Segment）

因为 TCP 是基于字节流传输的