

Lab2:配置Web服务器，分析HTTP交互过程

实验项目：搭建Web服务器、制作简单Web页面、分析HTTP交互过程

专业：物联网工程

学号：2213244

姓名：刘玉菡

一、实验目的

- 熟悉Web服务器的搭建流程。
- 掌握使用HTML、CSS构建简单Web页面的方法。
- 了解并分析HTTP请求与响应的交互过程。
- 使用Wireshark捕获并分析网络数据包，学习HTTP协议的工作机制。

二、实验环境




- 操作系统：Windows11
- Web服务器：IIS (Internet Information Services)
- 开发工具：Visual Studio 2022 (VS2022)
- 页面设计语言：HTML
- 网络抓包工具：Wireshark

三、实验步骤

1. 搭建Web服务器

- 启用IIS：在Windows的“控制面板 > 程序 > 启用或关闭Windows功能”中启用 **Internet Information Services (IIS)**。
- 配置网站：
 - 在IIS管理器中，选择“Default Web Site”并设定端口为8081（避免与系统占用的端口冲突）。
 - 将HTML页面及相关资源（LOGO图片、音频文件等）放入IIS根目录（`C:\inetpub\wwwroot`）。

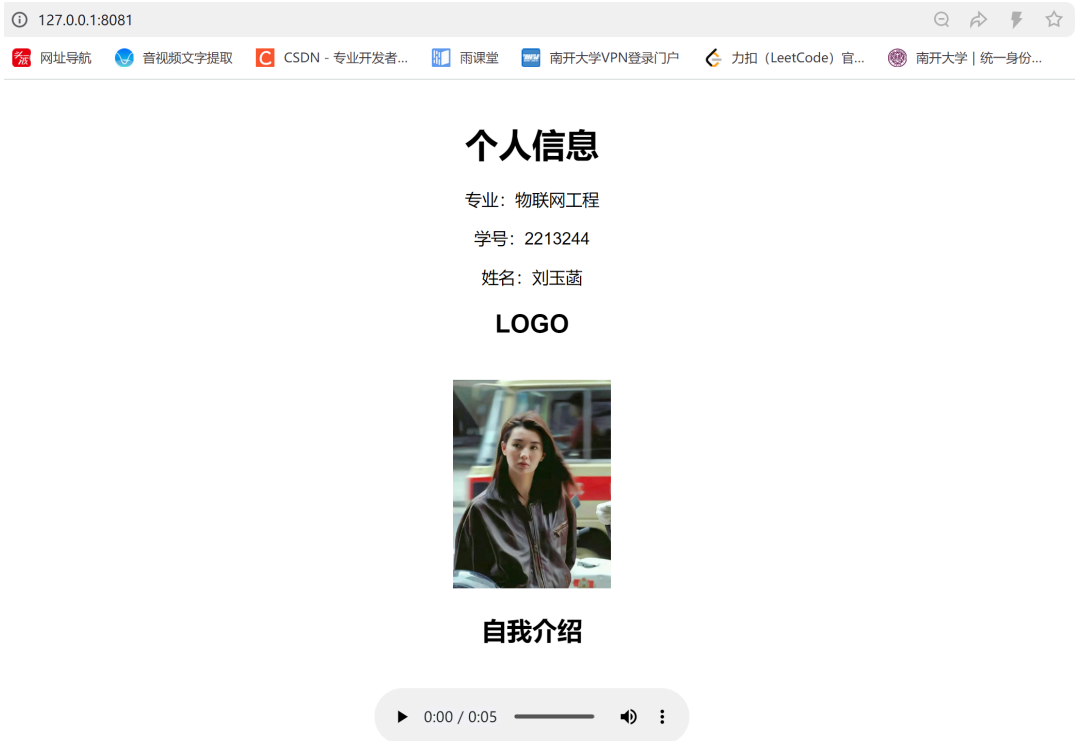
此电脑 > Windows-SSD (C:) > inetpub > wwwroot

名称	日期
 favicon	2024/10/28 23:25
 index	2024/10/28 23:28
 intro	2024/10/28 20:58
 logo	2024/10/28 16:46

2. 制作简单的Web页面

- 页面内容：

- HTML文件包含个人信息（专业、学号、姓名）、个人LOGO以及一段自我介绍音频。
- 页面代码不在此赘述，详情请看该报告同目录下index.html文件。
- 打开浏览器，输入 `http://127.0.0.1:8081` 得到如下界面。

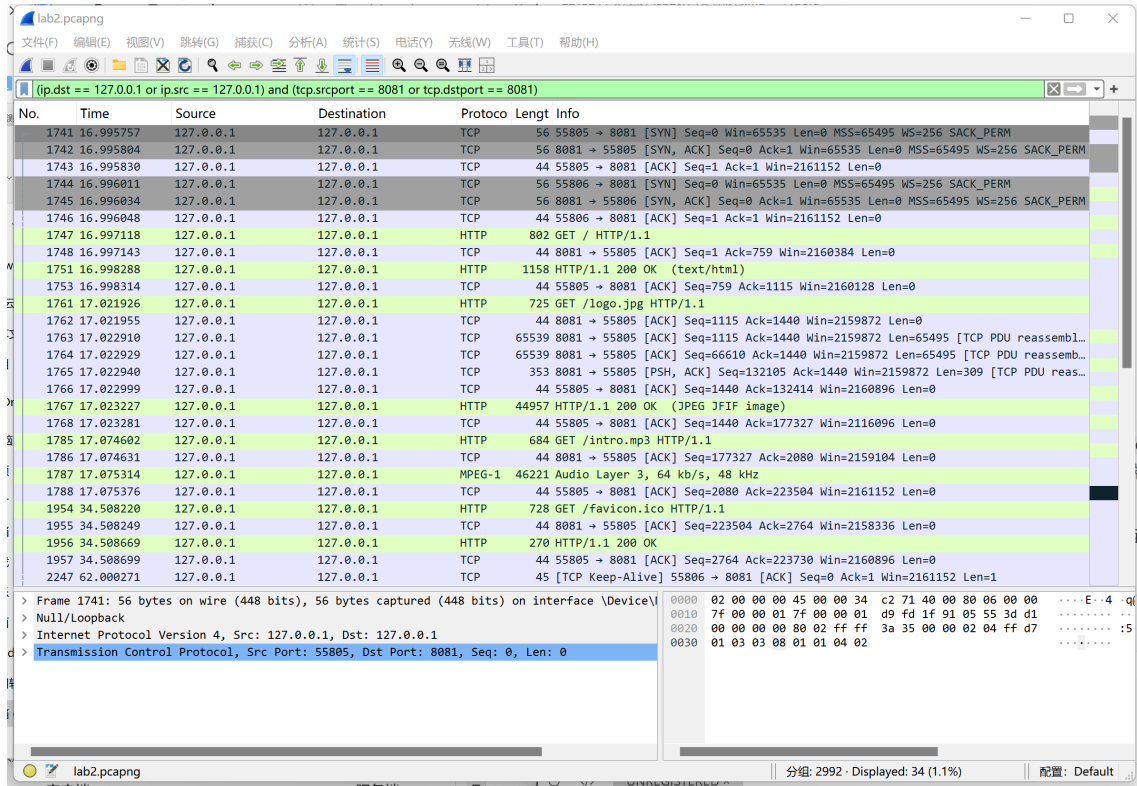


- 页面资源：

- **logo.jpg**：个人LOGO图片，放在服务器根目录。
- **intro.mp3**：自我介绍的音频文件，放在服务器根目录。
- **favicon.ico**：网页图标文件，用于显示在浏览器标签栏。

3. 使用Wireshark捕获交互过程

- 打开Wireshark并选择“Adapter for loopback traffic capture”接口。
- 设置过滤器 (ip.dst == 127.0.0.1 or ip.src == 127.0.0.1) and (tcp.srcport == 8081 or tcp.dstport == 8081) , 只显示8081端口的HTTP流量。
- 访问页面后，Wireshark开始捕获浏览器与Web服务器之间的HTTP交互。运行一段时间得到如下结果。



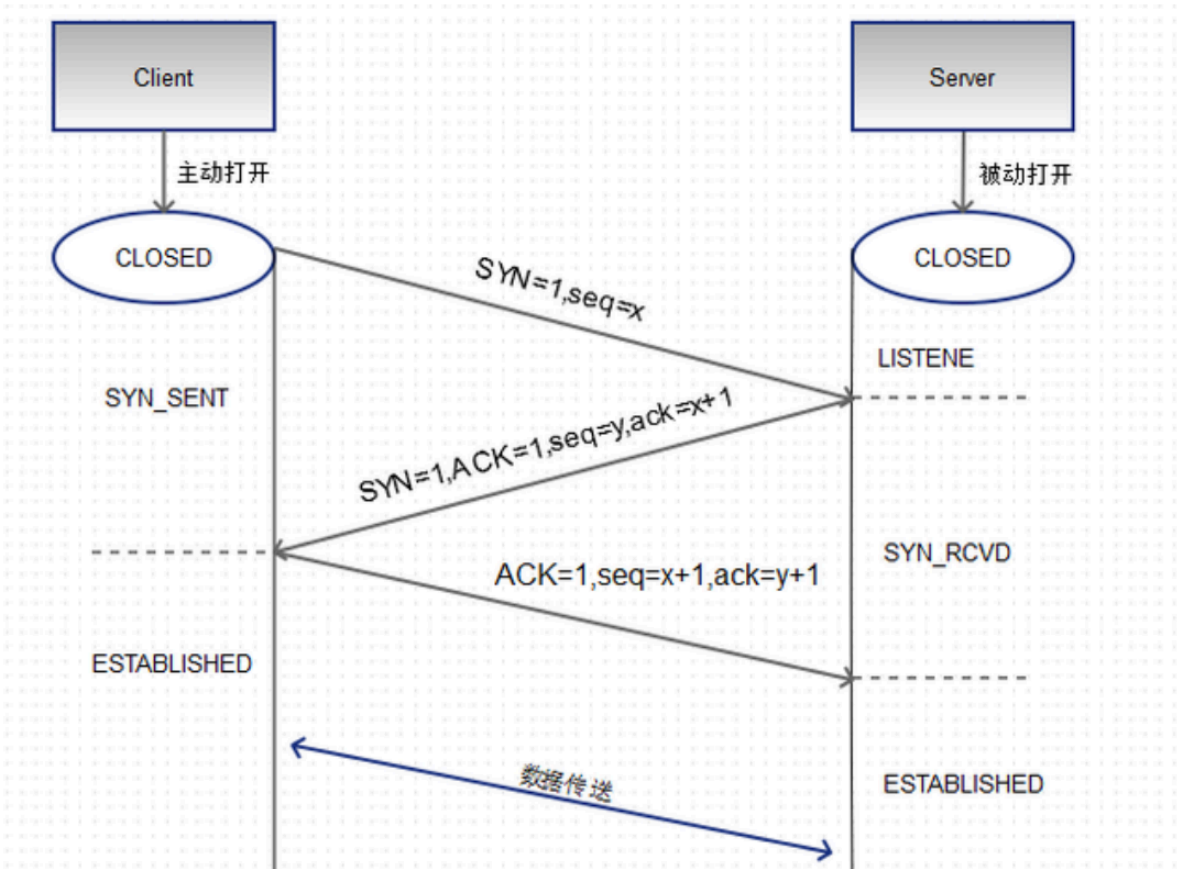
四、实验结果

1. TCP三次握手建立连接

TCP 三次握手的过程解决以下三个问题

1. 要是每一方都能确知对方的存在
2. 要允许双方协商一些参数(如窗口最大值，是否使用窗口扩大选项以及时间戳选项等)
3. 能够对运输实体资源(如缓冲大小、连接表中的项目等)进行分配

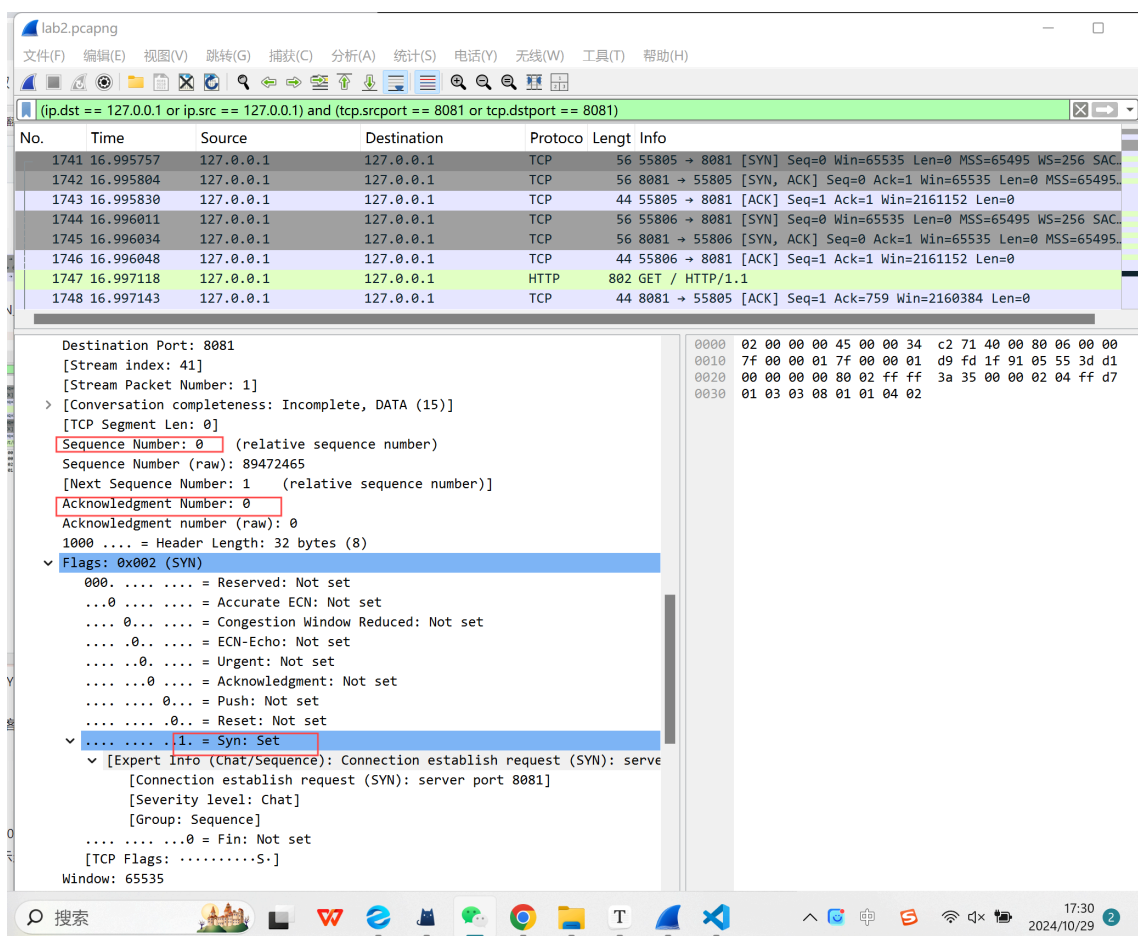
下图展示了连接建立的三次握手过程：



在wireshark中可以看到三次握手的过程：

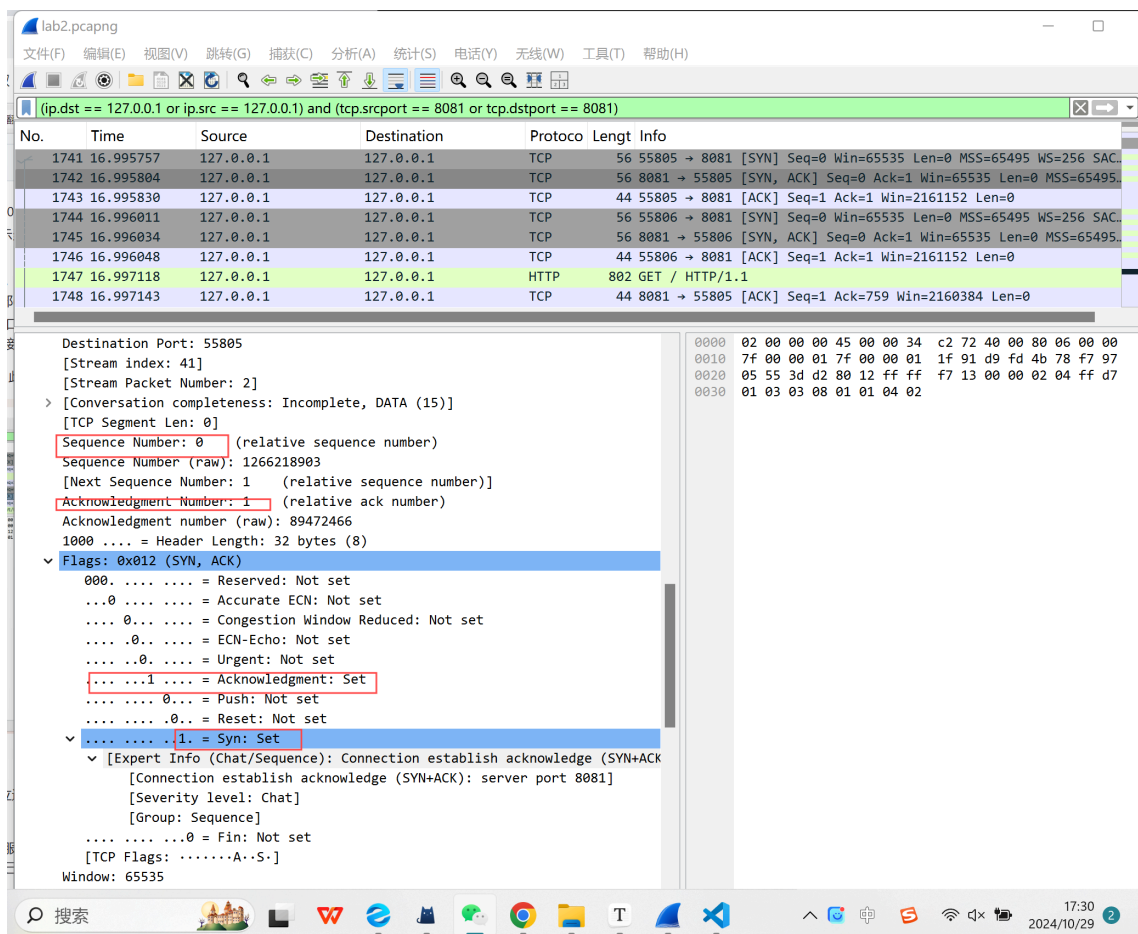
1741	16.995757	127.0.0.1	127.0.0.1	TCP	56 55805 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1742	16.995804	127.0.0.1	127.0.0.1	TCP	56 8081 → 55805 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1743	16.995830	127.0.0.1	127.0.0.1	TCP	44 55805 → 8081 [ACK] Seq=1 Ack=1 Win=2161152 Len=0

- **初始状态：**客户端处于 `closed` (关闭) 状态，服务器处于 `listen` (监听) 状态（有地方说也处于关闭状态）
- **第一次握手：**建立连接时，客户端发送 `SYN` 包 [`syn=x`] 到服务器，并进入 `SYN_Sent` 状态，表示请求与服务器建立连接，等待服务器确认。报文 `SYN = 1` 同步序列号和初始化序列号 `seq = x` 发送给服务端。这个步骤的作用是让客户端告诉服务器它准备好通信，并发送自己的初始序列号。



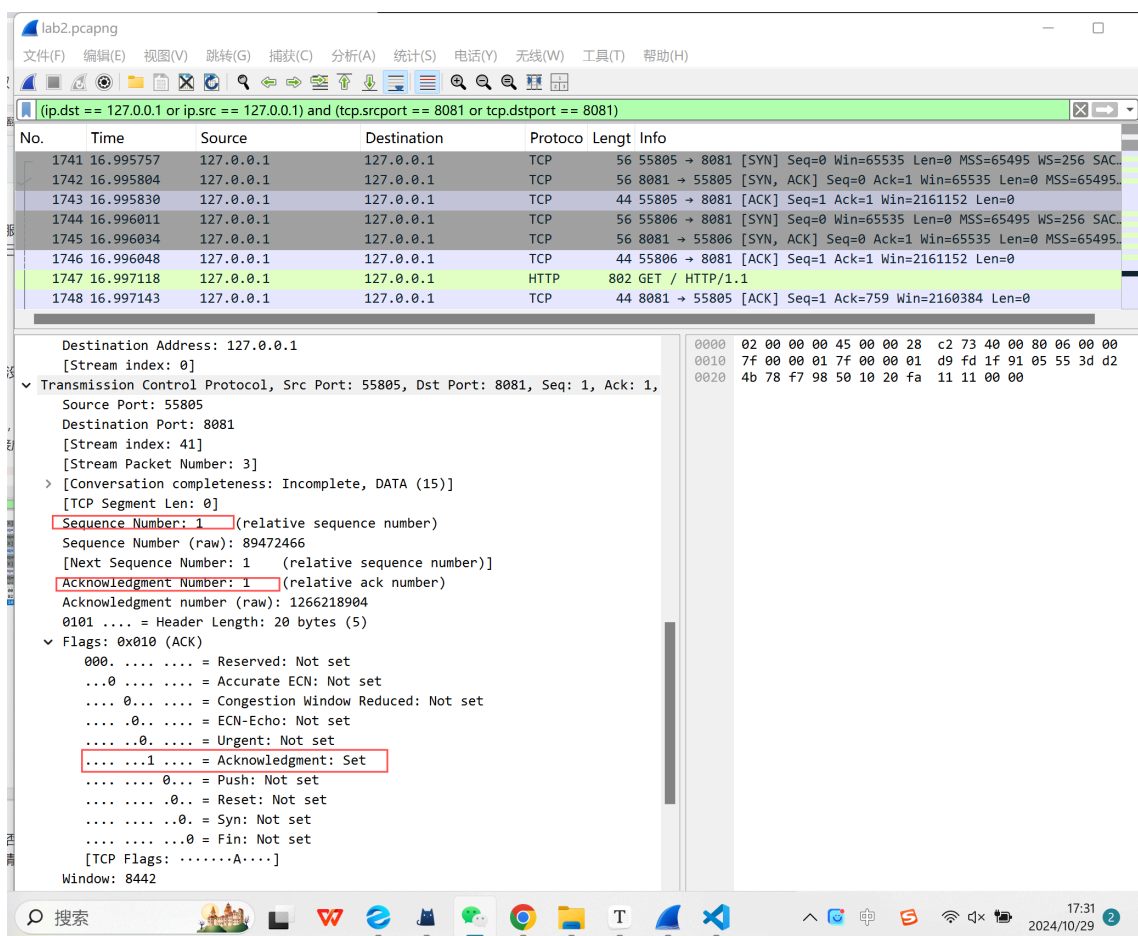
在第1741行的抓包数据中，可以看到客户端向服务器发送的 SYN 包，用于发起连接请求。主要参数如下：

- 源端口：55805（客户端发起）
 - 目标端口：8081（服务器监听端口）
 - 标志位：SYN 设置为 1
 - 序列号：Sequence Number = 0
 - 确认号：Acknowledgment Number = 0（此时不需要确认，因为是初始请求）
- 第二次握手：**服务端受到 SYN 请求报文之后，如果同意连接，会以自己的同步序列号 SYN = 1、初始化序列号 seq = y 和确认序列号（期望下次收到的数据包）ack = x + 1 以及确认号 ACK = 1 报文作为应答，服务器为 SYN_Receive 状态。这一步确保服务器接收到客户端的请求，同时将自己的连接意图传达给客户端。



在第1742行的抓包数据中，服务器响应客户端的连接请求，发送 SYN，ACK 包，表示接受连接请求并回复确认。主要参数如下：

- **源端口**：8081（服务器响应）
 - **目标端口**：55805（客户端端口）
 - **标志位**：SYN 和 ACK 都设置为 1
 - **序列号**：Sequence Number = 0
 - **确认号**：Acknowledgment Number = 1（确认客户端的序列号 0 + 1）
- **第三次握手**：客户端接收到服务端的 SYN + ACK 之后，知道下次可以发送了下一序列的数据包了，然后发送同步序列号 $ack = y + 1$ 和数据包的序列号 $seq = x + 1$ 以及确认号 $ACK = 1$ 确认包作为应答，客户端转为 established 状态。服务器收到客户端的应答报文后，也进入 ESTABLISHED 状态，正式建立连接。三次握手完成后，双方可以进行数据传输。



在第1743行的抓包数据中，客户端发送 ACK 包以确认连接，完成三次握手过程。主要参数如下：

- 源端口：55805（客户端）
- 目标端口：8081（服务器）
- 标志位：ACK 设置为 1
- 序列号：Sequence Number = 1（客户端的下一个序列号）
- 确认号：Acknowledgment Number = 1（确认服务器的序列号 0 + 1）

2. HTTP请求与响应

在连接建立后，客户端向服务器发送了4个HTTP请求，包括HTML页面、图片、音频文件和favicon图标。典型的 HTTP 请求由请求行（如 GET）、请求头和可选的消息体组成，服务器接收到请求后会返回 HTTP 响应，包括状态行（如 200 OK）、响应头和消息体。每种请求类型（如 GET、POST）对应不同的操作，GET 请求用于获取资源，POST 请求用于提交数据。在本实验中，主要观察了 GET 请求的发送和服务器的响应过程。

1747	16.997118	127.0.0.1	127.0.0.1	HTTP	802 GET / HTTP/1.1	← 请求
1748	16.997143	127.0.0.1	127.0.0.1	TCP	44 8081 → 55805 [ACK] Seq=1 Ack=759 Win=2160384 Len=0	
1751	16.998288	127.0.0.1	127.0.0.1	HTTP	1158 HTTP/1.1 200 OK (text/html)	← 响应
1753	16.998314	127.0.0.1	127.0.0.1	TCP	44 55805 → 8081 [ACK] Seq=759 Ack=1115 Win=2160128 Len=0	
1761	17.021926	127.0.0.1	127.0.0.1	HTTP	725 GET /logo.jpg HTTP/1.1	← 请求
1762	17.021955	127.0.0.1	127.0.0.1	TCP	44 8081 → 55805 [ACK] Seq=1115 Ack=1440 Win=2159872 Len=0	
1763	17.022910	127.0.0.1	127.0.0.1	TCP	65539 8081 → 55805 [ACK] Seq=1115 Ack=1440 Win=2159872 Len=65495 [TCP PDU reassembl...	
1764	17.022929	127.0.0.1	127.0.0.1	TCP	65539 8081 → 55805 [ACK] Seq=66610 Ack=1440 Win=2159872 Len=65495 [TCP PDU reasemb...	
1765	17.022940	127.0.0.1	127.0.0.1	TCP	353 8081 → 55805 [PSH, ACK] Seq=132105 Ack=1440 Win=2159872 Len=309 [TCP PDU reas...	
1766	17.022999	127.0.0.1	127.0.0.1	TCP	44 55805 → 8081 [ACK] Seq=1440 Ack=132414 Win=2160896 Len=0	
1767	17.023227	127.0.0.1	127.0.0.1	HTTP	44957 HTTP/1.1 200 OK (JPEG JFIF image)	← 响应
1768	17.023281	127.0.0.1	127.0.0.1	TCP	44 55805 → 8081 [ACK] Seq=1440 Ack=177327 Win=2116096 Len=0	
1785	17.074602	127.0.0.1	127.0.0.1	HTTP	684 GET /intro.mp3 HTTP/1.1	← 请求
1786	17.074631	127.0.0.1	127.0.0.1	TCP	44 8081 → 55805 [ACK] Seq=177327 Ack=2080 Win=2159104 Len=0	
1787	17.075314	127.0.0.1	127.0.0.1	MPEG-1	46221 Audio Layer 3, 64 kb/s, 48 kHz	← 响应
1788	17.075376	127.0.0.1	127.0.0.1	TCP	44 55805 → 8081 [ACK] Seq=2080 Ack=223504 Win=2161152 Len=0	
1954	34.508220	127.0.0.1	127.0.0.1	HTTP	728 GET /favicon.ico HTTP/1.1	← 请求
1955	34.508249	127.0.0.1	127.0.0.1	TCP	44 8081 → 55805 [ACK] Seq=223504 Ack=2764 Win=2158336 Len=0	
1956	34.508669	127.0.0.1	127.0.0.1	HTTP	270 HTTP/1.1 200 OK	← 响应

以下是每个请求的详细分析：

- HTML页面请求：

- **请求**：在 **第1747行**，客户端发起 `GET / HTTP/1.1` 请求以获取HTML页面内容。
- **响应**：在 **第1751行**，服务器返回 `HTTP/1.1 200 OK`，Content-Type 为 `text/html`，表明页面内容正常加载。
- **LOGO图片请求**：
 - **请求**：在 **第1761行**，客户端请求 `GET /logo.jpg`。
 - **响应**：在 **第1767行**，服务器返回 `HTTP/1.1 200 OK`，Content-Type 为 `image/jpeg`，表示图片资源加载成功。
- **音频文件请求**：
 - **请求**：在 **第1785行**，客户端请求 `GET /intro.mp3`。
 - **响应**：在 **第1787行**，服务器返回 `HTTP/1.1 200 OK`，Content-Type 为 `audio/mpeg`，说明音频文件成功加载。
- **favicon.ico请求**：
 - **请求**：在 **第1754行**，客户端请求 `GET /favicon.ico`。
 - **响应**：在 **第1756行**，服务器返回 `HTTP/1.1 200 OK`，Content-Type 为 `image/x-icon`，表明favicon图标加载成功，用于显示在浏览器标签栏上。

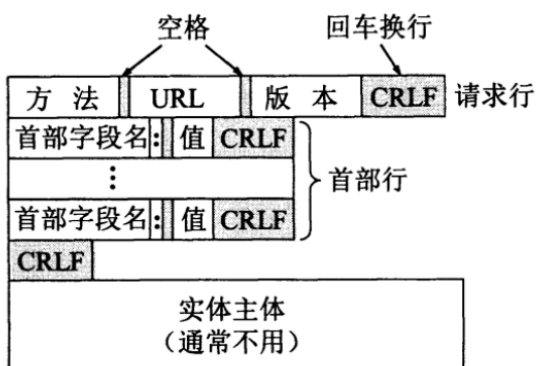
(一) HTTP报文结构

HTTP有两类报文：

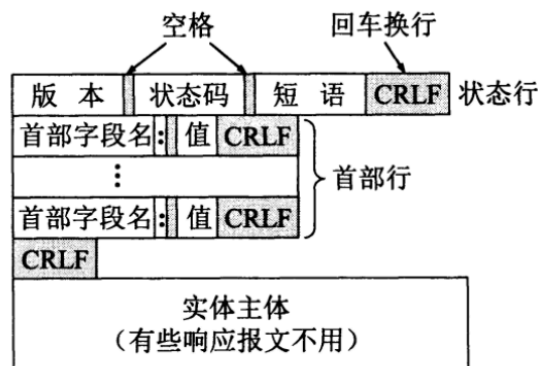
- 请求报文——从客户向服务器发送请求报文
- 响应报文——从服务器到客户的回答

HTTP请求报文和响应报文都是由三个部分组成的：

- **开始行**，用于区分是请求报文还是响应报文。在请求报文中的开始行叫做请求行(Request-Line)，而在响应报文中的开始行叫做状态行(Status-Line)。在开始行的三个字段之间都以空格分隔开，最后的“CR”和“LF”分别代表“回车”和“换行”。
- **首部行**，用来说明浏览器、服务器或报文主体的一些信息。首部可以有好几行，但也可以不使用。在每一个首部行中都有首部字段名和它的值，每一行在结束的地方都要有“回车”和“换行”。整个首部行结束时，还有一空行将首部行和后面的实体主体分开。
- **实体主体(entity body)**，在请求报文中一般都不用这个字段，而在响应报文中也可能没有这个字段。



(a) 请求报文



(b) 响应报文

(二) HTTP请求报文的方法

请求报文的第一行“请求行”只有三个内容，即**方法**，请求资源的URL，以及 HTTP协议的版本。

请求报文中常用的几种方法如下：

方法	作用
GET	获取资源
POST	传输实体主体
PUT	上传文件
DELETE	删除文件
HEAD	和GET方法类似，但只返回报文首部，不返回报文实体主体部分
PATCH	对资源进行部分修改
OPTIONS	查询指定的URL支持的方法
CONNECT	要求用隧道协议连接代理，用来进行环回测试的请求报文
TRACE	服务器会将通信路径返回给客户端，用于代理服务器

为了方便记忆，可以将PUT、DELETE、POST、GET理解为客户端对服务端的增删改查。

- PUT：上传文件，向服务器添加数据，可以看作增
- DELETE：删除文件
- POST：传输数据，向服务器提交数据，对服务器数据进行更新。
- GET：获取资源，查询服务器资源

(三) HTTP相应报文的状态码

HTTP响应报文的状态行包括三项内容，即 HTTP的版本，**状态码**，以及解释状态码的简单短语。

下面三种状态行在响应报文中是经常见到的：

HTTP/1.1 202 Accepted	{接受}
HTTP/1.1 400 Bad Request	{错误的请求}
Http/1.1 404 Not Found	{找不到}

状态码(Status-Code)负责表示客户端 HTTP 请求的返回结果、标记服务器端 的处理是否正常、通知出现的错误等工作。状态码(Status-Code)都是三位数字的，分为5大类。这5大类的状态码都是以不同的数字开头的。

	类别	原因短语
1xx	Informational（信息性状态码）	接收的请求正在处理
2xx	Success（成功状态码）	请求正常处理完毕
3xx	Redirection（重定向状态码）	需要进行附加操作以完成请求
4xx	Client Error（客户端错误状态码）	服务器无法处理请求
5xx	Server Error（服务器错误状态码）	服务器处理请求出错

让我们对第一轮 http请求与响应过程进行分析：

1.客户端向服务器发送请求

The image shows a Wireshark packet capture analysis of a network traffic. The top pane displays a list of packets, with packet 1747 highlighted. The middle pane shows the details of the selected packet, and the bottom pane shows the raw packet data in hexadecimal and ASCII.

Packet List:

No.	Time	Source	Destination	Protocol	Length	Info
1741	16.995757	127.0.0.1	127.0.0.1	TCP	56	55805 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK
1742	16.995804	127.0.0.1	127.0.0.1	TCP	56	8081 → 55805 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=6549
1743	16.995830	127.0.0.1	127.0.0.1	TCP	44	55805 → 8081 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
1744	16.996011	127.0.0.1	127.0.0.1	TCP	56	55806 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK
1745	16.996034	127.0.0.1	127.0.0.1	TCP	56	8081 → 55806 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=6549
1746	16.996048	127.0.0.1	127.0.0.1	TCP	44	55806 → 8081 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
1747	16.997118	127.0.0.1	127.0.0.1	HTTP	802	GET / HTTP/1.1
1748	16.997143	127.0.0.1	127.0.0.1	TCP	44	8081 → 55805 [ACK] Seq=1 Ack=759 Win=2160384 Len=0
1751	16.998288	127.0.0.1	127.0.0.1	HTTP	1158	HTTP/1.1 200 OK (text/html)

Packet Details:

- [Conversation completeness: Incomplete, DATA (15)]
- [TCP Segment Len: 758]
- Sequence Number: 1 (relative sequence number)
- Sequence Number (raw): 89472466
- [Next Sequence Number: 759 (relative sequence number)]
- Acknowledgment Number: 1 (relative ack number)
- Acknowledgment number (raw): 1266218904
- 0101 = Header Length: 20 bytes (5)
- ▼ Flags: 0x018 (PSH, 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
- ...1 = Push: Set
- ...0 = Reset: Not set
- ...0 = Syn: Not set
- ...0 = Fin: Not set
- [TCP Flags:AP...]
- Window: 8442
- [Calculated window size: 2161152]
- [Window size scaling factor: 256]
- Checksum: 0x0c2a [unverified]
- [Checksum Status: Unverified]
- Urgent Pointer: 0
- ▼ [Timestamps]

Raw Data:

```
0000 02 00 00 00 45 00 03 1e c2 77 40 00 80 06 00 00
0010 7f 00 00 01 7f 00 00 01 d9 fd 1f 91 05 55 3d d
0020 4b 78 f7 98 50 18 20 fa 0c 2a 00 00 47 45 54 2
0030 2f 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 7
0040 3a 20 31 32 37 2e 30 2e 30 2e 31 3a 38 30 38 3
0050 0d 0a 43 6f 6e 6e 65 63 74 69 6f 6e 6e 3a 20 6b 6
0060 65 70 2d 61 6c 69 76 65 0d 0a 73 65 63 2d 63 6
0070 2d 75 61 3a 20 22 43 68 72 6f 6d 69 75 6d 22 3
0080 76 3d 22 39 22 2c 20 22 4e 6f 74 3f 41 5f 42 7
0090 61 6e 64 22 3b 76 3d 22 38 22 0d 0a 73 65 63 2
00a0 63 68 2d 75 61 2d 6d 6f 62 69 6c 65 3a 20 3f 3
00b0 0d 0a 73 65 63 2d 63 68 2d 75 61 2d 70 6c 61 7
00c0 66 6f 72 6d 3a 20 22 57 69 6e 64 6f 77 73 22 0
00d0 0a 55 70 67 72 61 64 65 2d 49 6e 73 65 63 75 7
00e0 65 2d 52 65 71 75 65 73 74 73 3a 20 31 0d 0a 5
00f0 73 65 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69 6
0100 6c 61 2f 35 2e 30 20 28 57 69 6e 64 6f 77 73 2
0110 4e 54 20 31 30 2e 30 3b 20 57 69 6e 36 34 3b 2
0120 78 36 34 29 20 41 70 70 6c 65 57 65 62 4b 69 7
0130 2f 35 33 37 2e 33 36 20 28 4b 48 54 4d 4c 2c 2
0140 6c 69 6b 65 20 47 65 63 6b 6f 29 20 43 68 72 6
0150 6d 65 2f 31 30 39 2e 30 2e 30 2e 30 20 53 61 6
0160 61 72 69 2f 35 33 37 2e 33 36 20 53 4c 42 72 6
0170 77 73 65 72 2f 39 2e 30 2e 33 2e 35 32 31 31 2
0180 53 4c 42 43 68 61 6e 2f 31 30 35 0d 0a 41 63 6
0190 65 70 74 3a 20 74 65 78 74 2f 68 74 6d 6c 2c 6
01a0 70 70 6c 69 63 61 74 69 6f 6e 2f 78 68 74 6d 6
01b0 2b 78 6d 6c 2c 61 70 70 6c 69 63 61 74 69 6f 6
01c0 2f 78 6d 6c 3b 71 3d 30 2e 39 2c 69 6d 61 67 6
01d0 2f 61 76 69 66 2c 69 6d 61 67 65 2f 77 65 62 7
01e0 2c 69 6d 61 67 65 2f 61 70 6e 67 2c 2a 2f 2a 3
01f0 71 3d 30 2e 38 2c 61 70 70 6c 69 63 61 74 69 6
0200 6e 2f 73 69 67 6e 65 64 7d 65 68 63 68 61 6e 6
```

- 第 1747 行: 客户端向服务器发送 GET / HTTP/1.1 请求。

- 源端口: 55805
- 目标端口: 8081
- 标志位: PSH, ACK (立即推送并等待确认)
- 序列号: 1
- 确认号: 1
- 数据长度: 758 字节

2.服务器端回复 ACK 表示收到请求

The image shows a Wireshark packet capture window titled 'lab2.pcapng'. The filter bar at the top is set to '(ip.dst == 127.0.0.1 or ip.src == 127.0.0.1) and (tcp.srcport == 8081 or tcp.dstport == 8081)'. The packet list on the left shows several packets, with packet 1748 highlighted. The packet details pane on the right shows the structure of the selected packet (No. 1748, Time 16.997143, Source 127.0.0.1, Destination 127.0.0.1, Protocol TCP, Length 44). The packet is a TCP segment from 8081 to 55805, with Seq=1, Ack=759, Win=2160384, and Len=0. The flags field shows 'ACK' set. The packet bytes pane on the right shows the raw data of the packet.

No.	Time	Source	Destination	Protocol	Length	Info
1741	16.995757	127.0.0.1	127.0.0.1	TCP	56	55805 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SA
1742	16.995804	127.0.0.1	127.0.0.1	TCP	56	8081 → 55805 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=6549
1743	16.995830	127.0.0.1	127.0.0.1	TCP	44	55805 → 8081 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
1744	16.996011	127.0.0.1	127.0.0.1	TCP	56	55806 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SA
1745	16.996034	127.0.0.1	127.0.0.1	TCP	56	8081 → 55806 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=6549
1746	16.996048	127.0.0.1	127.0.0.1	TCP	44	55806 → 8081 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
1747	16.997118	127.0.0.1	127.0.0.1	HTTP	802	GET / HTTP/1.1
1748	16.997143	127.0.0.1	127.0.0.1	TCP	44	8081 → 55805 [ACK] Seq=1 Ack=759 Win=2160384 Len=0
1751	16.998288	127.0.0.1	127.0.0.1	HTTP	1158	HTTP/1.1 200 OK (text/html)

Packet 1748 details:

- [Conversation completeness: Incomplete, DATA (15)]
- [TCP Segment Len: 0]
- Sequence Number: 1 (relative sequence number)
- Sequence Number (raw): 1266218904
- [Next Sequence Number: 1 (relative sequence number)]
- Acknowledgment Number: 759 (relative ack number)
- Acknowledgment number (raw): 89473224
- 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: 8439
- [Calculated window size: 2160384]
- [Window size scaling factor: 256]
- Checksum: 0x0e1e [unverified]
- [Checksum Status: Unverified]
- Urgent Pointer: 0
- [Timestamps]

- 第 1748 行：服务器端发送 ACK 包，以确认收到客户端的请求。
 - 源端口：8081
 - 目标端口：55805
 - 标志位：ACK
 - 序列号：1
 - 确认号：759
 - 数据长度：0（只有确认，没有数据）

3.服务器向客户端发送响应报文

lab2.pcapng

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

(ip.dst == 127.0.0.1 or ip.src == 127.0.0.1) and (tcp.srcport == 8081 or tcp.dstport == 8081)

No.	Time	Source	Destination	Protocol	Length	Info
1741	16.995757	127.0.0.1	127.0.0.1	TCP	56	55805 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK
1742	16.995804	127.0.0.1	127.0.0.1	TCP	56	8081 → 55805 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=6549
1743	16.995830	127.0.0.1	127.0.0.1	TCP	44	55805 → 8081 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
1744	16.996011	127.0.0.1	127.0.0.1	TCP	56	55806 → 8081 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK
1745	16.996034	127.0.0.1	127.0.0.1	TCP	56	8081 → 55806 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=6549
1746	16.996048	127.0.0.1	127.0.0.1	TCP	44	55806 → 8081 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
1747	16.997118	127.0.0.1	127.0.0.1	HTTP	802	GET / HTTP/1.1
1748	16.997143	127.0.0.1	127.0.0.1	TCP	44	8081 → 55805 [ACK] Seq=1 Ack=759 Win=2160384 Len=0
1751	16.998288	127.0.0.1	127.0.0.1	HTTP	1158	HTTP/1.1 200 OK (text/html)

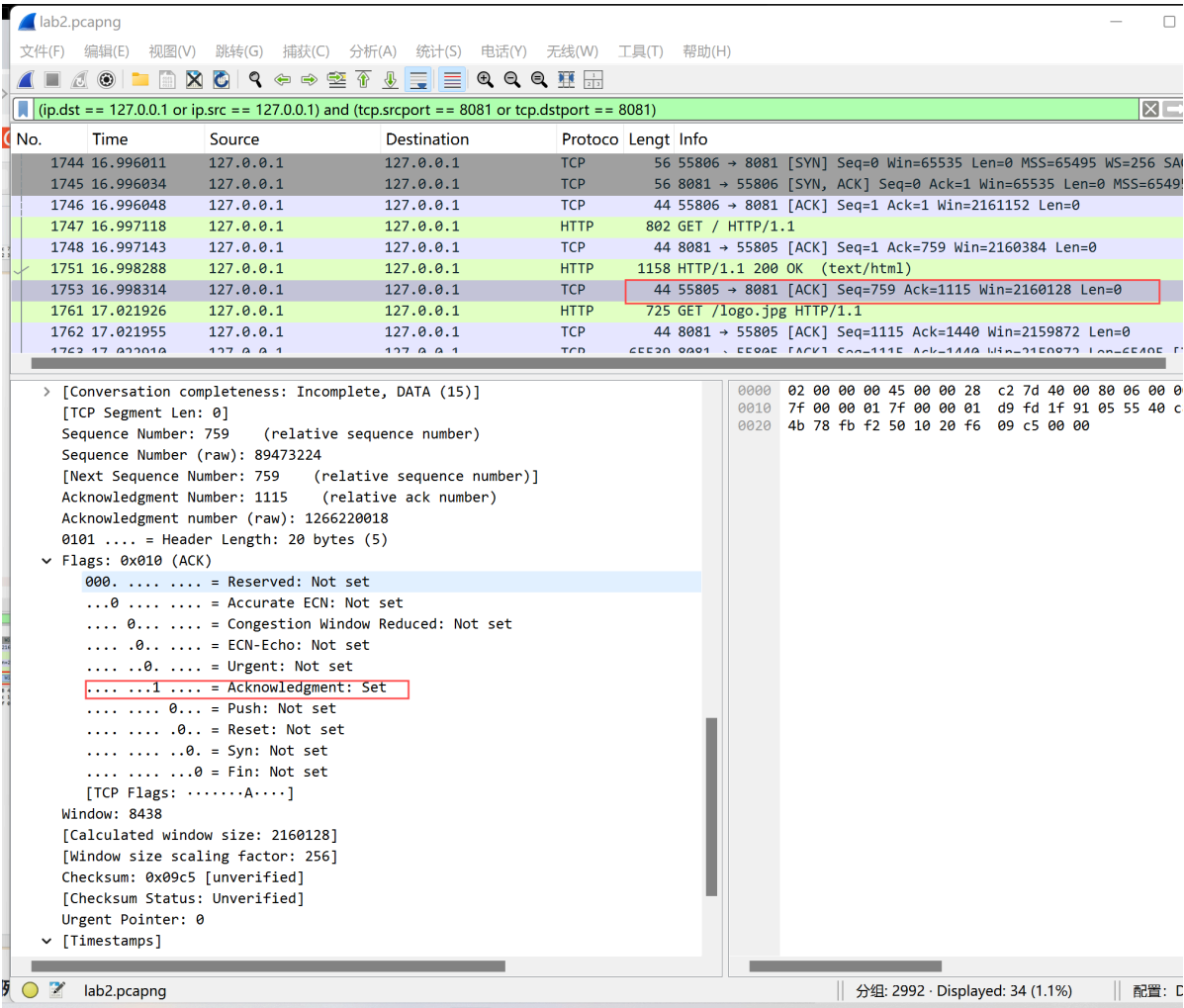
> [Conversation completeness: Incomplete, DATA (15)]
[TCP Segment Len: 1114]
Sequence Number: 1 (relative sequence number)
Sequence Number (raw): 1266218904
[Next Sequence Number: 1115 (relative sequence number)]
Acknowledgment Number: 759 (relative ack number)
Acknowledgment number (raw): 89473224
0101 = Header Length: 20 bytes (5)
▼ Flags: 0x018 (PSH, 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
...1 = Push: Set
...0 = Reset: Not set
...0 = Syn: Not set
...0 = Fin: Not set
[TCP Flags:AP...]
Window: 8439
[Calculated window size: 2160384]
[Window size scaling factor: 256]
Checksum: 0xdf36 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
▼ [Timestamps]

0000 02 00 00 00 45 00 04 82 c2 7b 40 00 80 06 00 00
0010 7f 00 00 01 7f 00 00 01 1f 91 d9 fd 4b 78 f7 9
0020 05 55 40 c8 50 18 20 f7 df 36 00 00 48 54 54 5
0030 2f 31 2e 31 20 32 30 30 20 4f 4b 0d 0a 43 6f 6
0040 74 65 6e 74 2d 54 79 70 65 3a 20 74 65 78 74 2
0050 68 74 6d 6c 0d 0a 4c 61 73 74 2d 4d 6f 64 69 6
0060 69 65 64 3a 20 4d 6f 6e 2c 20 32 38 20 4f 63 7
0070 20 32 30 32 34 20 31 35 3a 32 38 3a 32 34 20 4
0080 4d 54 0d 0a 41 63 63 65 70 74 2d 52 61 6e 67 6
0090 73 3a 20 62 79 74 65 73 0d 0a 45 54 61 67 3a 2
00a0 22 35 34 64 38 34 33 37 34 65 32 39 64 62 31 3
00b0 30 22 0d 0a 53 65 72 76 65 72 3a 20 4d 69 63 7
00c0 6f 73 6f 66 74 2d 49 49 53 2f 31 30 2e 30 0d 0
00d0 44 61 74 65 3a 20 4d 6f 6e 2c 20 32 38 20 4f 6
00e0 74 20 32 30 32 34 20 31 35 3a 33 39 3a 31 36 2
00f0 47 4d 54 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6
0100 67 74 68 3a 20 38 39 30 0d 0a 0d 0a 3c 21 44 4
0110 43 54 59 50 45 20 68 74 6d 6c 3e 0d 0a 3c 68 7
0120 6d 6c 20 6c 61 6e 67 3d 22 7a 68 2d 43 4e 22 3
0130 0d 0a 0d 0a 3c 68 65 61 64 3e 0d 0a 20 20 20 2
0140 3c 6d 65 74 61 20 63 68 61 72 73 65 74 3d 22 5
0150 54 46 2d 38 22 3e 0d 0a 20 20 20 20 3c 74 69 7
0160 6c 65 3e e4 b8 aa e4 ba ba e4 b8 bb e9 a1 b5 3
0170 2f 74 69 74 6c 65 3e 0d 0a 20 20 20 20 3c 6c 6
0180 6e 6b 20 72 65 6c 3d 22 69 63 6f 6e 22 20 68 7
0190 65 66 3d 22 2f 66 61 76 69 63 6f 6e 2e 69 63 6
01a0 22 20 74 79 70 65 3d 22 69 6d 61 67 65 2f 78 2
01b0 69 63 6f 6e 22 3e 0d 0a 20 20 20 20 3c 73 74 7
01c0 6c 65 3e 0d 0a 20 20 20 20 20 20 20 20 62 6f 6
01d0 79 20 7b 0d 0a 20 20 20 20 20 20 20 20 20 20 2
01e0 20 66 6f 6e 74 2d 66 61 6d 69 6c 79 3a 20 41 7
01f0 69 61 6c 2c 20 73 61 6e 73 2d 73 65 72 69 66 3
0200 0d 0a 20 20 20 20 20 20 20 20 20 20 20 20 6d 6

lab2.pcapng 分组: 2992 · Displayed: 34 (1.1%) 配置: C

- 第 1751 行：服务器向客户端发送 HTTP/1.1 200 OK(text/html) 响应报文，包含主页内容。
 - 源端口：8081
 - 目标端口：55805
 - 标志位：PSH, ACK
 - 序列号：1
 - 确认号：759
 - 数据长度：1114 字节

4.客户端回复 ACK 表示收到响应



- 第 1753 行：客户端发送 ACK 包，确认已成功接收到服务器的响应。
 - 源端口：55805
 - 目标端口：8081
 - 标志位：ACK
 - 序列号：759
 - 确认号：1115
 - 数据长度：0（只有确认，没有数据）

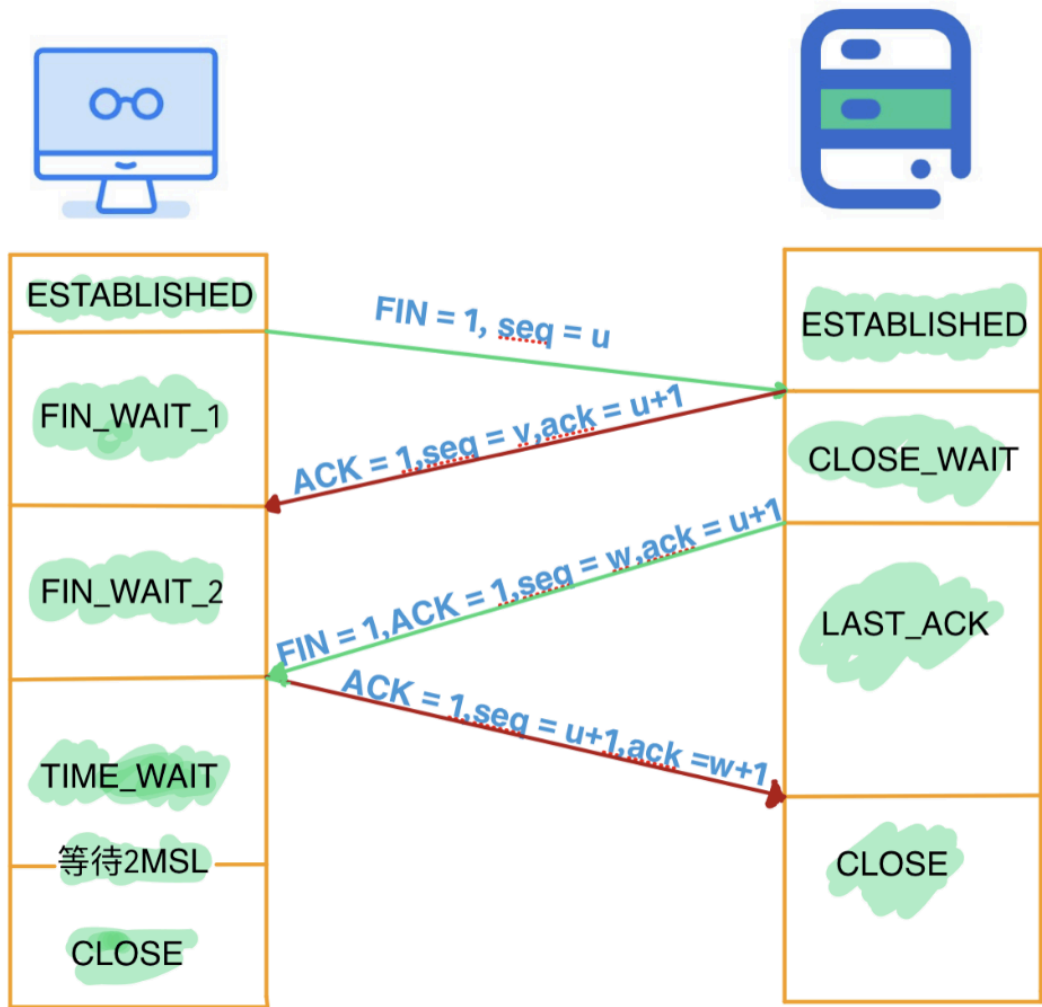
客户端的 ACK 包表示服务器发送的响应已经成功到达，第一轮请求和响应过程完成。

3. TCP四次挥手关闭连接

在所有HTTP请求完成后，客户端和服务端通过四次挥手来关闭连接，确保资源释放和连接关闭：

- 首先客户端要关闭连接,先把报文中标志位 FIN 置为1，然后向服务端发送FIN报文表示要关闭连接，之后客户端进入 FIN_WAIT 为 1 状态，但仍然可以接收数据。
- 服务端收到客户端发来的FIN报文之后,内核会自动回复一个 ACK 给客户端，之后服务端进入 CLOSED_WAIT 状态。表示接收到客户端的断开请求，但未准备好立即关闭。
- 等待服务端进程调用 close 函数，也就是等待服务端处理完数据之后，服务端在给客户端发送FIN 报文表示请求断开连接，之后服务端进入 LAST_ACK 状态。

- 客户端收到服务端发来的 FIN 报文之后,会回复最后一个 ACK 报文, 客户端进入 TIME_WAIT 状态, 2MSL 后,进入 closed 状态 (注意只有主动关闭连接的一方才会有 TIME_WAIT 状态)。
- 服务端接收到客户端的最后一个 ACK 报文后, 就会进入 closed 状态,关闭连接。
- 如果四次挥手过程中发生报文丢失, TCP 会通过超时重传机制进行恢复。例如, 如果客户端发送的 FIN 报文丢失, 客户端将等待超时后重传, 直到收到服务器的 ACK。同样, 若服务器的 FIN 丢失, 客户端会在 TIME_WAIT 状态中重新发送 ACK。这种机制保证了即使报文丢失, 连接仍能可靠关闭。



在抓包过程中, 由于浏览器在请求时通常会自动设置 Keep-Alive, 这意味着浏览器会尝试保持 TCP 连接不断开, 以便复用连接传输多个请求。这种行为导致在浏览器请求中不容易捕获到四次挥手的过程。

2247	62.000271	127.0.0.1	127.0.0.1	TCP	45 [TCP Keep-Alive] 55806 → 8081 [ACK] Seq=0 Ack=1 Win=216115
2248	62.000289	127.0.0.1	127.0.0.1	TCP	56 [TCP Dup ACK 1745#1] 8081 → 55806 [ACK] Seq=1 Ack=1 Win=216115
2359	79.512797	127.0.0.1	127.0.0.1	TCP	45 [TCP Keep-Alive] 55805 → 8081 [ACK] Seq=2763 Ack=223730 Win=216115
2360	79.512818	127.0.0.1	127.0.0.1	TCP	56 [TCP Keep-Alive ACK] 8081 → 55805 [ACK] Seq=223730 Ack=2763 Win=216115
2609	107.002501	127.0.0.1	127.0.0.1	TCP	45 [TCP Keep-Alive] 55806 → 8081 [ACK] Seq=0 Ack=1 Win=216115
2611	107.002523	127.0.0.1	127.0.0.1	TCP	56 [TCP Keep-Alive ACK] 8081 → 55806 [ACK] Seq=1 Ack=1 Win=216115
2895	124.522819	127.0.0.1	127.0.0.1	TCP	45 [TCP Keep-Alive] 55805 → 8081 [ACK] Seq=2763 Ack=223730 Win=216115
2896	124.522834	127.0.0.1	127.0.0.1	TCP	56 [TCP Keep-Alive ACK] 8081 → 55805 [ACK] Seq=223730 Ack=2763 Win=216115

为了避免浏览器的 Keep-Alive 设置对实验的干扰, 我通过在终端中使用 curl 命令 (curl http://127.0.0.1:8081) 来手动发起 HTTP 请求, 这样可以强制关闭连接, 确保捕获到标准的四次挥手过程。

在wireshark中可以看到四次挥手的过程:

155	18.438045	127.0.0.1	127.0.0.1	TCP	44 50559 → 8081 [FIN, ACK] Seq=78 Ack=1115 Win=326144 Len=0
156	18.438059	127.0.0.1	127.0.0.1	TCP	44 8081 → 50559 [ACK] Seq=1115 Ack=79 Win=2161152 Len=0
157	18.438068	127.0.0.1	127.0.0.1	TCP	44 8081 → 50559 [FIN, ACK] Seq=1115 Ack=79 Win=2161152 Len=0
158	18.438089	127.0.0.1	127.0.0.1	TCP	44 50559 → 8081 [ACK] Seq=79 Ack=1116 Win=326144 Len=0

1.客户端发送 FIN, ACK

- **数据包 156**: 客户端 (端口 50559) 向服务器 (端口 8081) 发送 FIN, ACK 包, 请求关闭连接。
- **标志位**: FIN, ACK
- **序列号**: 78
- **确认号**: 1115

2.服务器确认客户端的关闭请求 ACK

- **数据包 157**: 服务器 (端口 8081) 发送 ACK 包, 确认收到客户端的关闭请求。
- **标志位**: ACK
- **序列号**: 1115
- **确认号**: 79

3.服务器发送 FIN, ACK, 请求关闭

- **数据包 157**: 服务器 (端口 8081) 向客户端发送 FIN, ACK 包, 请求关闭连接。
- **标志位**: FIN, ACK
- **序列号**: 1115
- **确认号**: 79

4.客户端确认服务器的关闭请求 ACK

- **数据包 158**: 客户端 (端口 50559) 发送 ACK 包, 确认收到服务器的关闭请求。
- **标志位**: ACK
- **序列号**: 79
- **确认号**: 1116

在实际操作中, 操作系统可能会优化连接关闭过程。为了提高效率, 服务器可能将 FIN 和 ACK 包合并, 导致在抓包中无法清晰地看到独立的第三步。即三次挥手。

五、问题与思考

1.为什么是三次握手而不是一次或两次?

为了防止已失效的连接请求报文段突然又传送到了服务端, 因而产生错误。如果此时客户端发送的延迟的握手信息服务器收到, 然后服务器进行响应, 认为客户端要和它建立连接, 此时客户端并没有这个意思, 但 server 却以为新的运输连接已经建立, 并一直等待 client 发来数据。这样, server 的很多资源就白白浪费掉了。

2.为什么断开连接需要四次挥手而不是三次或两次?

TCP 协议的四次挥手是为了确保连接双方可以独立、可靠地关闭自己的发送和接收通道。每一方都需要发送一个 FIN 报文和一个 ACK 报文, 以此来表明自己不再发送数据, 同时确认对方的关闭请求。客户端首先发送 FIN 表示不再发送数据, 服务器收到后回复 ACK, 并等待所有数据处理完毕后再发送 FIN 请求关闭。最后, 客户端发送 ACK 确认断开。

四次挥手避免了数据丢失风险, 因为服务器不能立即发送 FIN, 需等到所有数据发送完成后再关闭。如果减少为三次或两次挥手, 可能导致数据丢失或资源浪费。因此, 四次挥手设计保障了连接的可靠断开。

3.为什么需要 TIME_WAIT 状态?

- 防止旧的报文进入了相同的四元组连接中
- 能够保证被动关闭方正常关闭.

4.为什么四次挥手后，主动方需等待2MSL

在TCP协议中，主动关闭连接的一方在发送最后一个ACK报文后，会进入一个称为 `TIMEWAIT` 的状态，并在这个状态中等待2个最大报文生存时间（Maximum Segment Lifetime, MSL）后才最终关闭连接。这样做主要是基于以下几个原因：

- 确保最后一个 `ACK` 报文的到达：等待2MSL可以确保最后一个 `ACK` 报文能够到达被动关闭方。如果最后一个 `ACK` 报文在网络中丢失，被动关闭方会重新发送 `FIN` 报文。在 `TIME-WAIT` 状态中，主动关闭方能够重新发送 `ACK` 报文来响应重发的 `FIN` 报文。
- 避免旧数据干扰新连接：等待2MSL也可以确保该连接持续期间的所有报文都从网络中消失，防止这些旧报文在连接关闭后误导新的连接。`MSL` 是网络中任何报文可能存在的最长时间，2MSL可以确保报文在两个方向上的传播都已经完全结束。

5.TCP协议有什么改进空间

- **头部压缩与减少开销**：TCP 头部相对较大，尤其在小数据包（如 IoT）传输时显得浪费。使用头部压缩或简化的协议（如 QUIC）可以提高小数据包的传输效率。
 - **连接建立与断开速度优化**：TCP 的三次握手和四次挥手在高频连接创建和断开时存在较大延迟。新的协议如 QUIC，通过减少握手次数和改进连接管理来提升速度，尤其适合短连接和实时应用。
-