



南开大学
Nankai University

南开大学

计算机学院和网络空间安全学院

《计算机网络》实验报告

Lab02: 配置 Web 服务器 & 分析 HTTP 交互过程

姓名：陆皓喆

学号：2211044

专业：信息安全

指导教师：徐敬东、张建忠

2024 年 10 月 31 日

目录

1 实验要求	2
2 github 仓库	2
3 实验原理	2
3.1 HTTP 协议	2
3.1.1 HTTP 特性	2
3.1.2 HTTP 主要组成部分	3
3.1.3 HTTP1.0&HTTP1.1 的区别	4
3.1.4 HTTP&HTTPS 的区别	4
3.2 TCP 协议	5
3.3 三次握手	6
3.4 四次挥手	7
4 实验环境	8
4.1 Web 服务器	8
4.2 客户端	9
5 实验过程	9
5.1 网页设计	9
5.2 运行 Web 服务器	12
5.3 配置 WireShark 信息	12
5.4 在客户端访问网页	12
5.5 WireShark 抓包结果分析	14
5.5.1 三次握手	14
5.5.2 访问 html 文件	16
5.5.3 四次挥手	17
6 一些思考	20
7 遇到的问题 & 解决方案	21
8 心得体会	22

1 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。
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, 使用 Wireshark 过滤器使其只显示 HTTP 协议。
3. 现场演示。
4. 提交 HTML 文档、Wireshark 捕获文件和实验报告, 对 HTTP 交互过程进行详细说明。
5. 页面不要太复杂, 包含所要求的基本信息即可。使用 HTTP, 不要使用 HTTPS。

2 github 仓库

本次实验的有关代码和文件, 都已经上传至我的个人 github 中。
您可以通过访问[此链接](#)来查阅我的代码文件。

3 实验原理

3.1 HTTP 协议

HTTP 协议 (超文本传输协议) 是网络中用于传输网页数据的主要协议, 它规定了客户端 (通常是浏览器) 和服务器之间的通信规则和格式。

在本次实验中, 我们采用 HTTP1.1 协议作为我们的 Web 环境。

3.1.1 HTTP 特性

HTTP 协议是一个基于请求-响应模式的协议, 用于从 Web 服务器传输超文本到本地浏览器。以下是关于 HTTP 协议的几个关键点:

1. 请求-响应模型

HTTP 采用请求-响应模型进行通信, 即客户端向服务器发送请求, 服务器处理请求后返回响应。请求和响应都包含三个主要部分:

- 请求行 / 响应行: 描述请求方法、URL 和协议版本, 响应行包含状态码和状态描述;
- 头部信息: 携带请求或响应的元数据 (如内容类型、编码方式、缓存控制等);
- 消息主体: 包含实际的数据内容 (如 HTML 页面、图片、JSON 数据等)。

2. 状态无关性

HTTP 是无状态协议, 意味着每次请求都是独立的, 服务器不会记录之前的请求信息。为了实现会话管理 (如用户登录、购物车等), 通常使用 Cookie、Session 等机制来维持客户端状态。

3. 可扩展性 & 灵活性

HTTP 协议的头部字段提供了高度的扩展性, 允许传输的内容包括各种类型的信息。此外, HTTP 协议支持多种内容格式和编码, 使其成为一个灵活且易于扩展的协议。

3.1.2 HTTP 主要组成部分

HTTP 协议主要由请求部分和响应部分组成，下面一一进行介绍。

1. 请求

HTTP 请求消息由**请求行**、**请求头部**、**空行**、**请求体**四部分组成。

- **请求行 (Request Line):** 请求行包含三个元素：**请求方法**、**请求 URL** 和 **HTTP 协议版本**。请求方法表示客户端希望执行的操作，如 GET、POST、PUT 等；请求 URI 表示标识资源的路径，如 /index.html；HTTP 版本表示指定使用的 HTTP 协议版本，如 HTTP/1.1 或 HTTP/2。
- **请求头部 (Request Headers):** 请求头部包含多个字段，每个字段包括一个键值对，用于传递请求的附加信息，如内容类型、用户代理等。常见的请求头部字段包括：
 - **Host:** 指定请求的服务器主机名；
 - **User-Agent:** 标识客户端的信息（如浏览器类型）；
 - **Accept:** 告知服务器客户端可接受的内容类型；
 - **Content-Type:** 指定请求体的数据类型（在 POST 请求中常用）。
- **空行:** 请求头部结束后有一个空行，表示头部部分的结束，后续是请求体内容。
- **请求体 (Request Body):** 请求体包含了请求中要传递的数据，通常在 POST、PUT 等方法中使用。请求体数据格式可以是表单数据、JSON、XML 等，具体取决于 Content-Type 字段。

2. 响应

HTTP 响应消息由**状态行**、**响应头部**、**空行**、**响应体**四部分组成。

- **状态行 (Status Line):** 状态行包含 HTTP 版本、状态码和状态描述。其中包含：
 - **HTTP 版本:** 指定响应的 HTTP 协议版本；
 - **状态码:** 表示响应的结果，常见的状态码如 200（成功）、404（未找到）、500（服务器错误）；
 - **状态描述:** 对状态码的文字描述，通常与状态码意义对应（如“OK”或“Not Found”）。
- **响应头部 (Response Headers):** 响应头部包含多个字段，用于传递响应的元数据。常见的响应头部字段包括：
 - **Content-Type:** 响应内容的类型，如 text/html、application/json；
 - **Content-Length:** 响应体的字节长度；
 - **Server:** 服务器信息；
 - **Set-Cookie:** 设置客户端存储的 Cookie。
- **空行:** 响应头部结束后有一个空行，表示头部部分的结束，后续是响应体内容。
- **响应体 (Response Body):** 响应体包含了实际返回的数据内容，比如 HTML 页面、图片、JSON 数据等。响应体的格式由 Content-Type 字段指定，客户端根据内容类型进行解析。

3.1.3 HTTP1.0&HTTP1.1 的区别

搜索资料可得，两者具有以下这些区别：

特性	HTTP/1.0	HTTP/1.1
连接管理	短连接，每次请求建立新连接	长连接（Keep-Alive），支持连接复用
缓存机制	使用 Expires 控制缓存过期时间	增加 Cache-Control 和 ETag 等字段，提供更灵活的缓存控制
错误处理	支持有限的状态码	增加了更多状态码，如 100 Continue、409 Conflict 等
请求头和响应头	支持的头部字段较少	增加了更多头部字段，如 Host、Connection 和 Transfer-Encoding
带宽优化	无优化机制，所有请求头和响应头必须完整传输	支持内容压缩，如 Content-Encoding，减少带宽消耗
范围请求	不支持	支持 Range 请求，允许请求部分资源
安全性	基本的认证机制	新增 Upgrade 头支持向 HTTPS 迁移
URL 路径	使用绝对路径	强制要求包含 Host 头支持虚拟主机

3.1.4 HTTP&HTTPS 的区别

通过搜索可得，HTTP 和 HTTPS 的区别如下所示：

特性	HTTP	HTTPS
传输协议	超文本传输协议（Hypertext Transfer Protocol）	安全超文本传输协议（HTTP Secure），在 HTTP 基础上添加了 SSL/TLS 加密层
端口号	默认端口为 80	默认端口为 443
数据加密	数据未加密，以明文形式传输	数据通过 SSL/TLS 加密，保障传输的机密性和完整性
安全性	不提供安全保障，容易受到中间人攻击、窃听等威胁	提供身份验证、防止篡改和窃听，增强了安全性
证书要求	不需要数字证书	需要由权威机构签发的 SSL 证书来验证服务器身份
速度	因为无加密过程，通常比 HTTPS 稍快	因为包含加密和解密过程，传输速度可能略慢
SEO 优势	无 SEO 优势	搜索引擎（如 Google）对 HTTPS 网站有更高的排名优待

3.2 TCP 协议

TCP(传输控制协议) 是一种面向连接、可靠、字节流的传输层通信协议。它提供了一种保证数据完整性和按顺序到达的方式来发送数据。TCP 通过使用序号、确认、重传和流控制等机制来实现这些目标。

1. 源端口号 (Source Port)

源端口号为 16 位整数，表示发送方的应用程序端口。用于标识数据来源于哪一个应用进程。

2. 目标端口号 (Destination Port)

目标端口号为 16 位整数，表示接收方的应用程序端口。用于将数据发送到指定应用进程。

3. 序列号 (Sequence Number)

序列号为 32 位整数，用于标识数据段的位置。每一个字节在数据流中的位置都由序列号标识，确保数据能够按正确顺序重组。

4. 确认号 (Acknowledgment Number)

确认号为 32 位整数，表示接收方期待收到的下一个字节的序号。用于向发送方确认已接收到的数据，实现可靠性传输。

5. 数据偏移 (Data Offset)

数据偏移为 4 位字段，表示 TCP 报头的长度，以 32 位字（4 字节）为单位。该字段用于确定数据起始位置。

6. 保留字段 (Reserved)

保留字段为 6 位，当前未使用，全部置为 0。预留用于未来 TCP 协议扩展。

7. 标志位 (Flags)

标志位包含 6 个控制位，用于控制 TCP 连接的状态，常见标志位包括：

- (a) **URG**: 紧急标志，表明紧急指针字段有效。
- (b) **ACK**: 确认标志，表明确认号字段有效。
- (c) **PSH**: 推送标志，表明接收方应立即将数据传递给应用程序。
- (d) **RST**: 重置标志，表明连接出现问题，需要重置。
- (e) **SYN**: 同步标志，用于连接建立的三次握手过程。
- (f) **FIN**: 结束标志，用于连接断开时的四次挥手过程。

8. 窗口大小 (Window Size)

窗口大小为 16 位整数，表示接收方当前能够接收的最大数据量（字节数）。该字段用于流量控制，确保发送方不发送超过接收方处理能力的数据。

9. 校验和 (Checksum)

校验和为 16 位，用于校验 TCP 报文在传输过程中的完整性。校验和由发送方计算，接收方验证，用于发现数据传输中的错误。

10. 紧急指针 (Urgent Pointer)

紧急指针为 16 位整数，当 URG 标志被置位时，紧急指针表示当前数据中紧急数据的结束位置。该字段通常用于需要优先处理的数据。

11. 选项 (Options)

选项字段长度不定，用于支持扩展功能。常见选项包括最大报文段长度 (MSS)、时间戳等。选项字段的长度由数据偏移字段指定。

12. 数据 (Data)

数据字段包含了实际传输的数据内容，长度不定，具体取决于数据偏移字段和整个 TCP 报文长度。数据字段用于在 TCP 连接中传递实际的应用数据。

3.3 三次握手

TCP 三次握手主要目的是为了确保双方都准备好进行通信，以下是 TCP 握手的流程：

1. 第一次握手：客户端发送 SYN 包

客户端向服务器发送一个 SYN (同步序列编号) 包，用于请求建立连接。此时，客户端进入 **SYN-SENT** 状态，等待服务器的响应。SYN 包包含客户端的初始序列号 (Sequence Number)，用于后续数据传输中的确认。

2. 第二次握手：服务器发送 SYN-ACK 包

服务器收到客户端的 SYN 包后，若同意建立连接，则向客户端发送一个带有 SYN 和 ACK (确认) 标志的包，即 SYN-ACK 包。该包中包含服务器的初始序列号，同时确认了客户端的序列号。此时，服务器进入 **SYN-RECEIVED** 状态。

3. 第三次握手：客户端发送 ACK 包

客户端收到服务器的 SYN-ACK 包后，向服务器发送一个 ACK 包，表示确认收到服务器的回应，并且连接已建立。此时，客户端和服务器都进入 **ESTABLISHED** 状态，连接成功，可以开始数据传输。

完成以上三个步骤后，TCP 连接被成功建立，双方可以开始发送数据。

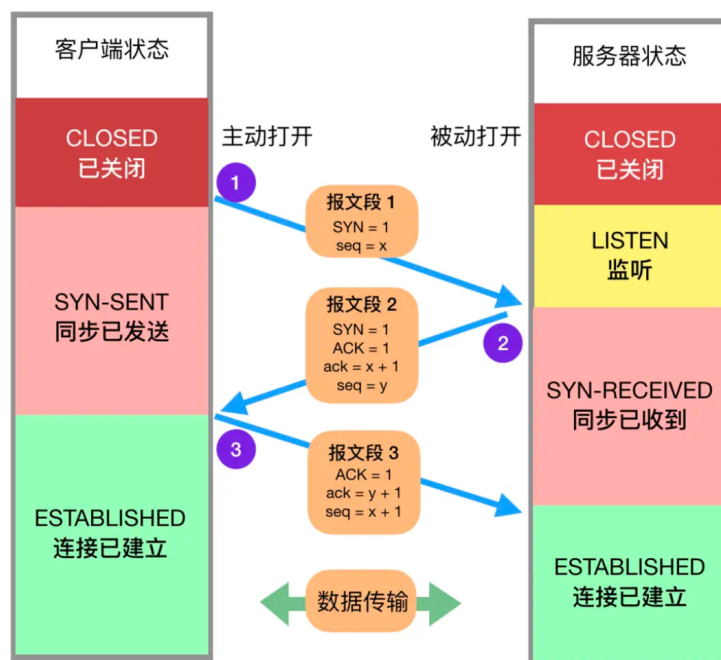


图 3.1: 三次握手

三次握手的主要目的是确认客户端和服务器的发送与接收能力。通过三次握手，双方可以：

- 确认彼此的发送和接收能力是否正常。
- 确保双方的初始序列号同步，为后续的数据传输奠定基础。
- 避免历史连接请求的干扰，例如旧的重复数据包影响当前连接的建立。

3.4 四次挥手

TCP 四次挥手是断开 TCP 连接的过程，包括以下步骤：

1. 第一次挥手：客户端发送 FIN 包

当客户端希望断开连接时，向服务器发送一个 FIN（结束）包，表示客户端不再发送数据。此时，客户端进入 **FIN-WAIT-1** 状态，等待服务器的响应。

2. 第二次挥手：服务器发送 ACK 包

服务器收到客户端的 FIN 包后，向客户端发送一个 ACK 包，确认已经接收到客户端的断开请求，但仍可能有数据要发送。此时，服务器进入 **CLOSE-WAIT** 状态，而客户端进入 **FIN-WAIT-2** 状态，继续等待服务器的 FIN 包。

3. 第三次挥手：服务器发送 FIN 包

当服务器完成所有数据的发送后，向客户端发送一个 FIN 包，表示服务器也准备断开连接。此时，服务器进入 **LAST-ACK** 状态，等待客户端的最终确认。

4. 第四次挥手：客户端发送 ACK 包

客户端收到服务器的 FIN 包后，向服务器发送一个 ACK 包，确认连接终止。此时，客户端进入 **TIME-WAIT** 状态，以确保服务器能够收到 ACK 包。经过一段时间后，客户端进入 **CLOSED** 状态，连接彻底关闭。服务器在收到 ACK 包后立即进入 **CLOSED** 状态。

完成以上四个步骤后，TCP 连接被成功断开。

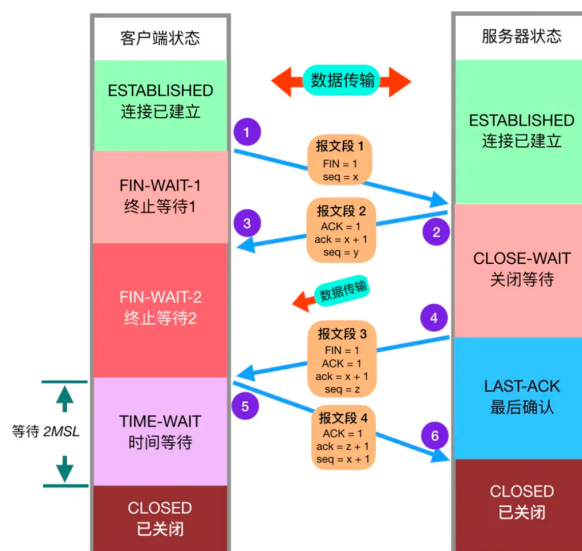


图 3.2: 四次挥手

四次挥手的主要目的是确保双方都能完整地接收已发送的数据，避免数据丢失或中断。通过四次挥手，双方可以：

- 确保剩余数据在连接断开前能被完整传输。
- 允许双方独立地关闭发送和接收通道，确保数据接收完整性。
- 防止连接意外中断，确保双方正确关闭连接状态。

4 实验环境

4.1 Web 服务器

我们选用了上一学期的《软件安全》课程中配置的 Web 环境，也就是使用 PHPnow 中的页面来作为我们的 Web 端的服务器。界面如图4.3所示：

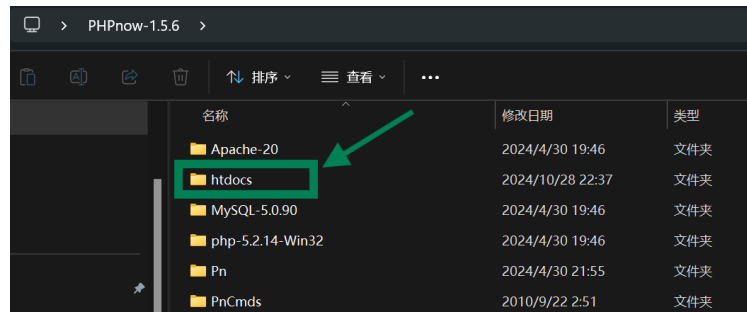


图 4.3: Web 环境配置

我们把我们的 html 文件存储在 htdocs 目录下即可，将图片和音频信息存储在该目录下的 src 文件夹中。

然后，我们将我们设计的 html 页面以及图片和音频文件均存放到了该目录中，如图4.4所示。

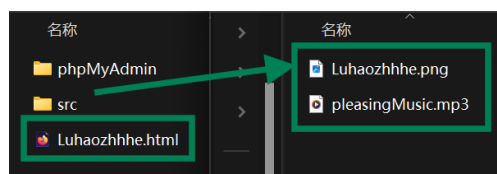


图 4.4: 文件存放情况

然后我们在主机终端输入 ipconfig，查看本机的 ipv4 地址，如图4.5所示：

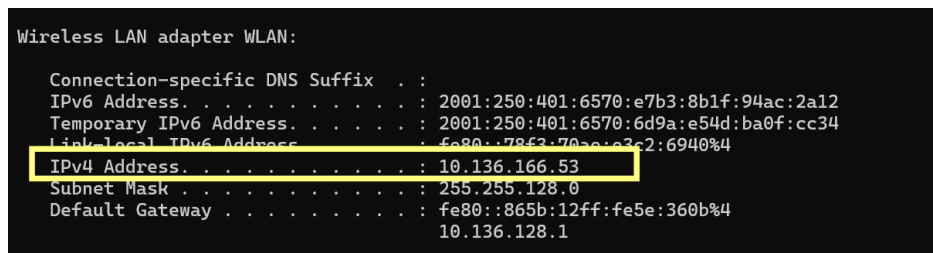
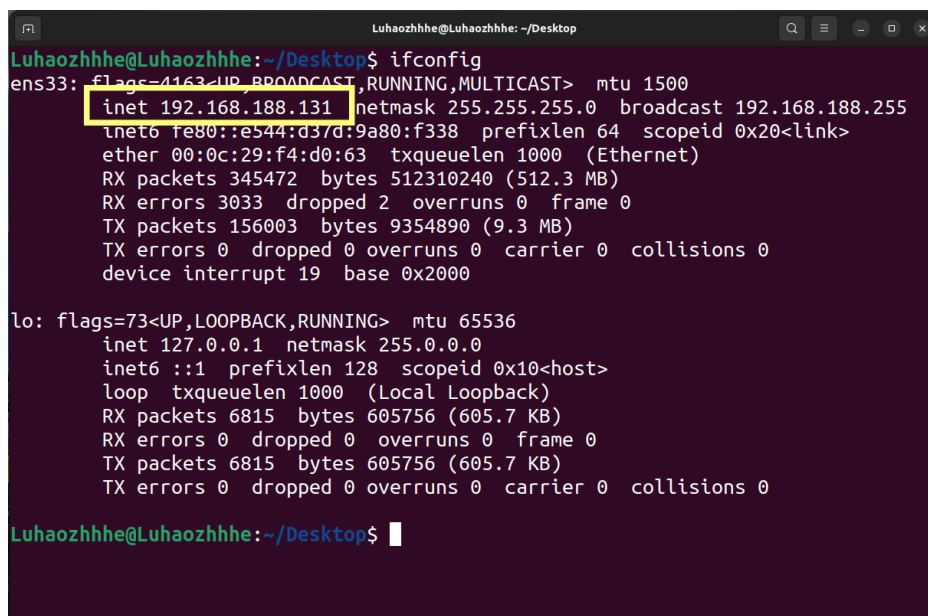


图 4.5: 服务器的 ip 地址

由上面的结果可以得出：我们服务器端的 ip 端口为 10.136.166.53。

4.2 客户端

本次实验，我们使用 ubuntu 虚拟机来模拟我们客户端的网络环境，虚拟机中选用 VMnet8，对应的 ip 地址如图4.6所示。



```
Luhaozhhe@Luhaozhhe: ~/Desktop
Luhaozhhe@Luhaozhhe:~/Desktop$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.188.131 netmask 255.255.255.0 broadcast 192.168.188.255
    inet6 fe80::e544:d37d:9a80:f338 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:f4:d0:63 txqueuelen 1000 (Ethernet)
    RX packets 345472 bytes 512310240 (512.3 MB)
    RX errors 3033 dropped 2 overruns 0 frame 0
    TX packets 156003 bytes 9354890 (9.3 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 19 base 0x2000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 6815 bytes 605756 (605.7 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6815 bytes 605756 (605.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

Luhaozhhe@Luhaozhhe:~/Desktop$
```

图 4.6: 客户端的 ip 端口

由上面的结果可以得出：我们的客户端 ip 端口号为 192.168.188.131。

5 实验过程

5.1 网页设计

首先，我们要完成我们网页的设计，由于该部分不是本次实验的重点，所以我简单完成了该部分的任务。

对应的 html 源码如下所示：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Welcome To Luhaozhhe's Home!</title>
7     <style>
8         body{
9             text-align: center;
10            /* beautiful font */
```

```
11 .hcqStyle1{color:hsl(184,80%,25%);text-shadow:0 0 1px
    ↪ currentColor,/*highlight*/-1px -1px 1px hsl(184,80%,50%),0 -1px
    ↪ 1px hsl(184,80%,55%),1px -1px 1px hsl(184,80%,50%),/*light
    ↪ shadow*/1px 1px 1px hsl(184,80%,10%),0 1px 1px
    ↪ hsl(184,80%,10%),-1px 1px 1px hsl(184,80%,10%),/*outline*/-2px
    ↪ -2px 1px hsl(184,80%,15%),-1px -2px 1px hsl(184,80%,15%),0 -2px
    ↪ 1px hsl(184,80%,15%),1px -2px 1px hsl(184,80%,15%),2px -2px 1px
    ↪ hsl(184,80%,15%),2px -1px 1px hsl(184,80%,15%),2px 0 1px
    ↪ hsl(184,80%,15%),2px 1px 1px hsl(184,80%,15%),-2px 0 1px
    ↪ hsl(184,80%,15%),-2px -1px 1px hsl(184,80%,15%),-2px 1px 1px
    ↪ hsl(184,80%,15%),/*dark shadow*/2px 2px 2px hsl(184,80%,5%),1px
    ↪ 2px 2px hsl(184,80%,5%),0 2px 2px hsl(184,80%,5%),-1px 2px 2px
    ↪ hsl(184,80%,5%),-2px 2px 2px hsl(184,80%,5%)}
12 }
13 .profile{
14     background: linear-gradient(rgba(255, 255, 255, 0.4), rgba(255, 255,
    ↪ 255, 0.4)), url(src/Luhaozhhe.png);
15     background-attachment: fixed;
16     background-repeat: no-repeat;
17     background-position: center;
18     background-size: cover;
19     height: 100vh;
20 }
21 audio{
22     float: right;
23 }
24 .text{
25     width: 300px;
26     height: 200px;
27     position: absolute;
28     left: 50%;
29     margin-left: -150px;
30     background-image: -webkit-linear-gradient(left,blue,#66ffff
    ↪ 10%,#cc00ff 20%,#CC00CC 30%, #CCCCFF 40%, #00FFFF 50%,#CCCCFF
    ↪ 60%,#CC00CC 70%,#CC00FF 80%,#66FFFF 90%,blue 100%);
31     -webkit-text-fill-color: transparent;
32     -webkit-background-clip: text;
33     -webkit-background-size: 200% 100%;
34     -webkit-animation: masked-animation 4s linear infinite;
35     font-size: 30px;
36 }
37
```

```
38     </style>
39 </head>
40 <body>
41     <div class="container">
42         <h1>Welcome To Luhaozhhe's Home!</h1>
43         
44         <h2> 个人信息 </h2>
45         <p><strong> 姓名 </strong>: 陆皓喆 </p>
46         <p><strong> 年龄 </strong>: 20 岁 </p>
47         <p><strong> 专业 </strong>: 信息安全 </p>
48         <p><strong> 学号 </strong>: 2211044</p>
49         <p><strong>github 主页 </strong>: https://github.com/Luhaozhhe</p>
50
51         <h2> 教育背景 </h2>
52         <ul>
53             <li><strong> 南开大学 </strong> - 网络空间安全学院 (2022.9-至今) </li>
54         </ul>
55
56         <h2> 专业技能&荣誉 </h2>
57         <ul>
58             <li><strong> 专业技能 </strong>: python、C++、MySQL、CTF-Crypto</li>
59             <li><strong>荣誉</strong>: 国家奖学金、2024
60                 ↳ 年京津冀信息安全攻防大赛一等奖等</li>
61         </ul>
62
63         <h2> 个人兴趣爱好 </h2>
64         <p> 热爱音乐，喜欢拍照;Enjoy Coding! </p>
65
66         <audio controls>
67             <source src="src/pleasingMusic.mp3" type="audio/mpeg">
68         </audio>
69     </div>
70 </body>
71 </body>
72 </html>
```

然后我们在服务器端访问该网址，界面如图5.7所示：



图 5.7: 个人页面展示

可以发现，我们的文字部分、图片头像部分和音频部分均可以正常播放。

5.2 运行 Web 服务器

我们只需要开启我们的 Web 服务器端口即可，由于我们在上学期的实验中已经开启了实验所需的端口信息，所以此处我们不再重复该步骤，直接进入下一步。

5.3 配置 WireShark 信息

首先，我们打开我们的 WireShark，首先需要选择主机和 NAT 模式的虚拟机所在的网络环境，即 VMware Network Adapter VMnet8。

由于 TCP 协议建立连接时候的三次握手和结束连接时候的四次挥手会是抓包分析的重点，为了能够同时捕获到主机和客户机对彼此发送的数据进行分析，需要让主机和客户机都轮流作为 src 和 dst。所以根据上述信息，我们设计了以下的过滤条件来进行抓包。

```
1 tcp.port==80&&
2 (((ip.src==10.136.166.53&&ip.host==192.168.188.131)|| (ip.src==192.168.188.131&&ip.h
  ↩  ost==10.136.166.53))
```

上面的过滤条件的含义为：即只显示 tcp 协议，以及源 IP 和目标 IP 是本机和虚拟机 (或虚拟机和本机)。

这样我们就准备好了，可以开始捕获对应的信息了！！

5.4 在客户端访问网页

我们在虚拟机中打开我们的服务器端的 ip，访问我们设计的页面，如图5.8所示。



图 5.8: 在客户端访问网页

我们发现，在本机上的 WireShark 中已经捕获到了我们所需要的信息，如图5.9和5.10所示。

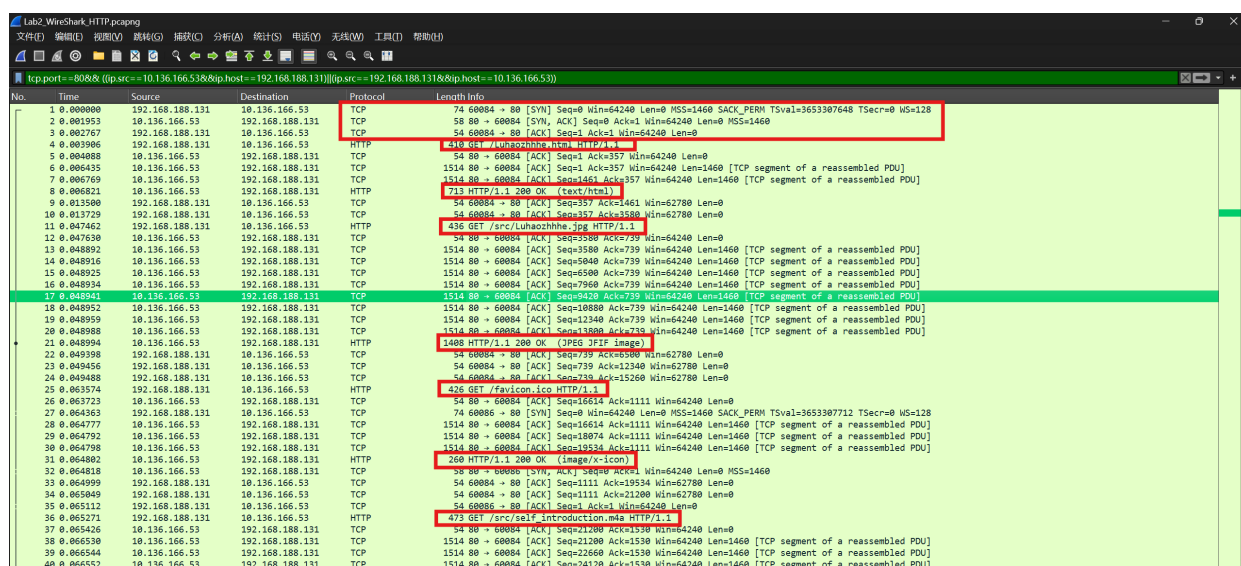


图 5.9: WireShark 捕获情况

No.	Time	Source	Destination	Protocol	Length Info
25	0.063574	192.168.188.131	10.136.166.53	HTTP	426 GET /favicon.ico HTTP/1.1
26	0.063723	10.136.166.53	192.168.188.131	TCP	54 60084 → 80 [ACK] Seq=16614 Ack=1111 Win=64240 Len=0
27	0.064363	192.168.188.131	10.136.166.53	TCP	74 60086 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3653307712 TSecr=0 WS=128
28	0.064777	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=16614 Ack=1111 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
29	0.064792	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=18074 Ack=1111 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
30	0.064798	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=19334 Ack=1111 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
31	0.064802	10.136.166.53	192.168.188.131	HTTP	260 HTTP/1.1 200 OK (image/x-icon)
32	0.064818	10.136.166.53	192.168.188.131	TCP	54 80 → 60086 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
33	0.064999	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1111 Ack=19334 Win=62780 Len=0
34	0.065049	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1111 Ack=21200 Win=62780 Len=0
35	0.065112	192.168.188.131	10.136.166.53	TCP	54 60086 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
36	0.065271	192.168.188.131	10.136.166.53	HTTP	473 GET /src/self_introduction.m4a HTTP/1.1
37	0.065426	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=22600 Ack=1330 Win=64240 Len=0
38	0.065530	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=21200 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
39	0.065544	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=22600 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
40	0.065552	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=24120 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
41	0.065557	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=25380 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
42	0.065562	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=27040 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
43	0.065567	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=28580 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
44	0.065574	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=29960 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
45	0.065579	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=31420 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
46	0.065583	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=32880 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
47	0.065589	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=34340 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
48	0.065594	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=35800 Ack=1330 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
49	0.065602	10.136.166.53	192.168.188.131	HTTP	1378 HTTP/1.1 206 Partial Content (text/plain)
50	0.065757	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1930 Ack=21200 Win=62780 Len=0
51	0.065782	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1530 Ack=27040 Win=62780 Len=0
52	0.065801	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1530 Ack=29960 Win=62780 Len=0
53	0.065875	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1530 Ack=38584 Win=62780 Len=0
54	0.065883	192.168.188.131	10.136.166.53	TCP	54 60086 → 80 [FIN, ACK] Seq=1 Ack=1 Win=64240 Len=0
55	0.067247	10.136.166.53	192.168.188.131	TCP	54 80 → 60086 [ACK] Seq=1 Ack=2 Win=64239 Len=0
56	0.067646	10.136.166.53	192.168.188.131	TCP	54 80 → 60086 [FIN, PSH, ACK] Seq=1 Ack=2 Win=64239 Len=0
57	0.068062	192.168.188.131	10.136.166.53	TCP	54 60086 → 80 [ACK] Seq=2 Ack=2 Win=64240 Len=0
58	0.220749	192.168.188.131	10.136.166.53	TCP	54 [TCP Keep-Alive] 60084 → 80 [ACK] Seq=1529 Ack=38584 Win=62780 Len=0
59	11.246487	192.168.188.131	10.136.166.53	TCP	54 [TCP Keep-Alive] 60084 → 80 [ACK] Seq=1529 Ack=38584 Win=62780 Len=0
60	11.246635	10.136.166.53	192.168.188.131	TCP	54 [TCP Keep-Alive] ACK 1 80 → 60084 [ACK] Seq=38584 Ack=1530 Win=64240 Len=0
61	15.068107	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [FIN, ACK] Seq=1530 Ack=38584 Win=62780 Len=0
62	15.068338	10.136.166.53	192.168.188.131	TCP	54 80 → 60084 [ACK] Seq=38584 Ack=1531 Win=64239 Len=0
63	15.068658	10.136.166.53	192.168.188.131	TCP	54 80 → 60084 [FIN, PSH, ACK] Seq=38584 Ack=1531 Win=64239 Len=0
64	15.069083	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1531 Ack=38585 Win=62780 Len=0

图 5.10: WireShark 捕获情况 (续)

5.5 WireShark 抓包结果分析

首先, 根据作业要求, 我们首先展示一下用 HTTP 来进行过滤得到的抓包情况:

No.	Time	Source	Destination	Protocol	Length Info
4	0.003906	192.168.188.131	10.136.166.53	HTTP	410 GET /Luhaozhhe.html HTTP/1.1
8	0.006821	10.136.166.53	192.168.188.131	HTTP	713 HTTP/1.1 200 OK (text/html)
11	0.047462	192.168.188.131	10.136.166.53	HTTP	436 GET /src/Luhaozhhe.jpg HTTP/1.1
21	0.048994	10.136.166.53	192.168.188.131	HTTP	1408 HTTP/1.1 200 OK (JPEG JFIF image)
25	0.063574	192.168.188.131	10.136.166.53	HTTP	426 GET /favicon.ico HTTP/1.1
31	0.064802	10.136.166.53	192.168.188.131	HTTP	260 HTTP/1.1 200 OK (image/x-icon)
36	0.065271	192.168.188.131	10.136.166.53	HTTP	473 GET /src/self_introduction.m4a HTTP/1.1
49	0.065602	10.136.166.53	192.168.188.131	HTTP	1378 HTTP/1.1 206 Partial Content (text/plain)

图 5.11: HTTP 筛选情况

我们发现, 抓包结果中成功展示了 GET 我们的 html、头像、系统界面、音频等内容!

这里需要进行一下说明, 我们的最后一条信息中的 206 消息, 说明我们的音频容量太大了, 所以实际上并没有完成所有内容的读取, 只导入了部分内容。这个我们在后续会进一步进行分析。

下面, 我们就对我们抓包得到的结果进行详细的分析:

由于我们页面中的照片、音频和 html 页面均为我们需要访问的页面, 所以此处我们对我们的三次握手、四次挥手和访问 html 文件、访问音频文件等来进行分析。

5.5.1 三次握手

如图5.12所示, 以下是我们得到的三次握手。

No.	Time	Source	Destination	Protocol	Length Info
1	0.000000	192.168.188.131	10.136.166.53	TCP	74 60084 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=3653307648 TSecr=0 WS=128
2	0.001953	10.136.166.53	192.168.188.131	TCP	58 80 → 60084 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
3	0.002767	192.168.188.131	10.136.166.53	TCP	54 60084 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0

图 5.12: 三次握手情况

首先是第一次握手。


```
> Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface \Device\NPF_{66D9EA02-6B1C-4568-978C-1272E8C8D9C7}, id 0
> Ethernet II, Src: VMware_f4:d0:63 (00:0c:29:f4:d0:63), Dst: VMware_f5:cc:52 (00:50:56:f5:cc:52)
> Internet Protocol Version 4, Src: 192.168.188.131, Dst: 10.136.166.53
> Transmission Control Protocol, Src Port: 60084, Dst Port: 80, Seq: 0, Len: 0
  Source Port: 60084
  Destination Port: 80
  [Stream index: 0]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 0]
  Sequence Number: 0 (relative sequence number)
  Sequence Number (raw): 2560016480
  [Next Sequence Number: 1 (relative sequence number)]
  Acknowledgment Number: 0 (relative sequence number)
  Acknowledgment number (raw): 0
  1010 .... = Header Length: 40 bytes (10)
  > 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
    > ....0... = Syn: Set
    ....0... = Fin: Not set
    [TCP Flags: .....S.]
  Window: 64240
  [Calculated window size: 64240]
  Checksum: 0xf467 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
  > [Timestamps]
```

图 5.13: 第一次握手分析

这是由客户端发送给服务端的包，前两行是源端口和目的端口，分别为 60084 和 80。客户端随机生成了一个初始的序列号，为 2560016480。这是我们的绝对序列号，对应的相对序列号为 0。

我们的 ACK 标志位没有置位，SYN 被置位，表明这是一次建立连接的请求，至此客户端进入 SYN-SENT 状态。

然后是第二次握手。

```
> Frame 2: 58 bytes on wire (464 bits), 58 bytes captured (464 bits) on interface \Device\NPF_{66D9EA02-6B1C-4568-978C-1272E8C8D9C7}, id 0
> Ethernet II, Src: VMware_f5:cc:52 (00:50:56:f5:cc:52), Dst: VMware_f4:d0:63 (00:0c:29:f4:d0:63)
> Internet Protocol Version 4, Src: 10.136.166.53, Dst: 192.168.188.131
> Transmission Control Protocol, Src Port: 80, Dst Port: 60084, Seq: 0, Ack: 1, Len: 0
  Source Port: 80
  Destination Port: 60084
  [Stream index: 0]
  [Conversation completeness: Complete, WITH_DATA (31)]
  [TCP Segment Len: 0]
  Sequence Number: 0 (relative sequence number)
  Sequence Number (raw): 1136010827
  [Next Sequence Number: 1 (relative sequence number)]
  Acknowledgment Number: 1 (relative ack number)
  Acknowledgment number (raw): 2560016481
  0110 .... = Header Length: 24 bytes (6)
  > 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
    > ....0... = Syn: Set
    ....0... = Fin: Not set
    [TCP Flags: .....A-S.]
  Window: 64240
  [Calculated window size: 64240]
  Checksum: 0xc13d [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > Options: (4 bytes), Maximum segment size
  > [Timestamps]
  > [SEQ/ACK analysis]
```

图 5.14: 第二次握手分析

这是服务器发送给客户端的包，源端口和目的端口分别为 80 和 60084。

服务端随机生成了一个初始的序列号，为 1136010827，为绝对序列号，对应的相对序列号为 0。

可以发现，我们的 ACK 也被置位了，确认号 (ack 字段) 的绝对值为第一次握手的 seq+1，即 $2560016480+1=2560016481$ ，相对值为 1。

我们还可以发现，SYN 被置位，表明服务器也建立了连接，至此服务器进入 SYN-RCVD 状态。

最后是第三次握手。

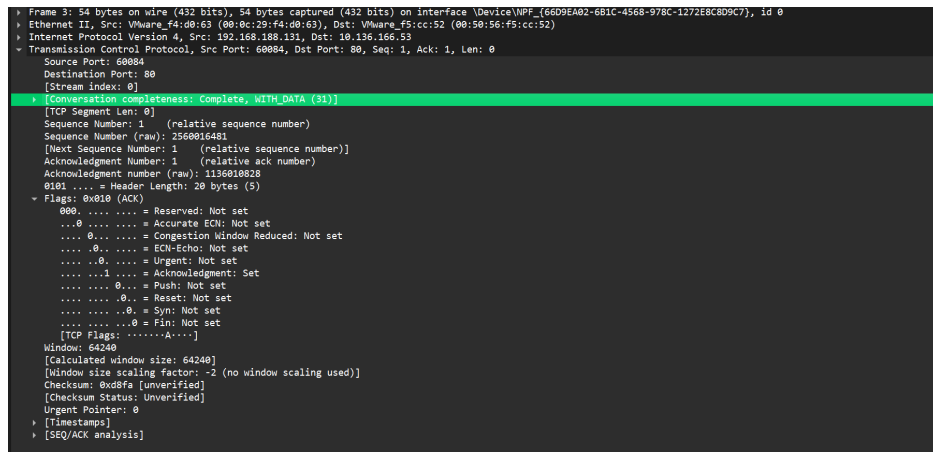


图 5.15: 第三次握手分析

这是客户端发给服务器的包，源端口和目的端口分别为 60084 和 80。

客户端的序列号为 1(相对客户端上一次发包 +1)。ACK 被置位，确认号 (ack 字段) 的绝对值为第二次握手服务端的 seq+1，即 $1136010827+1=1136010828$ ，相对值为 1。

至此，客户端进入 Established 状态，待服务端收到这个包后，也将进入这个状态，连接建立。

5.5.2 访问 html 文件

分析完我们开头的三次握手之后，我们继续分析我们的访问操作，也就是我们的 get 操作。

4	0.003906	192.168.188.131	10.136.166.53	HTTP	410 GET /Luhaozhhe.html HTTP/1.1
5	0.004088	10.136.166.53	192.168.188.131	TCP	54 80 → 60084 [ACK] Seq=1 Ack=357 Win=64240 Len=0
6	0.006435	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=1 Ack=357 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
7	0.006769	10.136.166.53	192.168.188.131	TCP	1514 80 → 60084 [ACK] Seq=1461 Ack=357 Win=64240 Len=1460 [TCP segment of a reassembled PDU]
8	0.006821	10.136.166.53	192.168.188.131	HTTP	713 HTTP/1.1 200 OK (text/html)

图 5.16: 访问 html 文件

查看其对应内容：

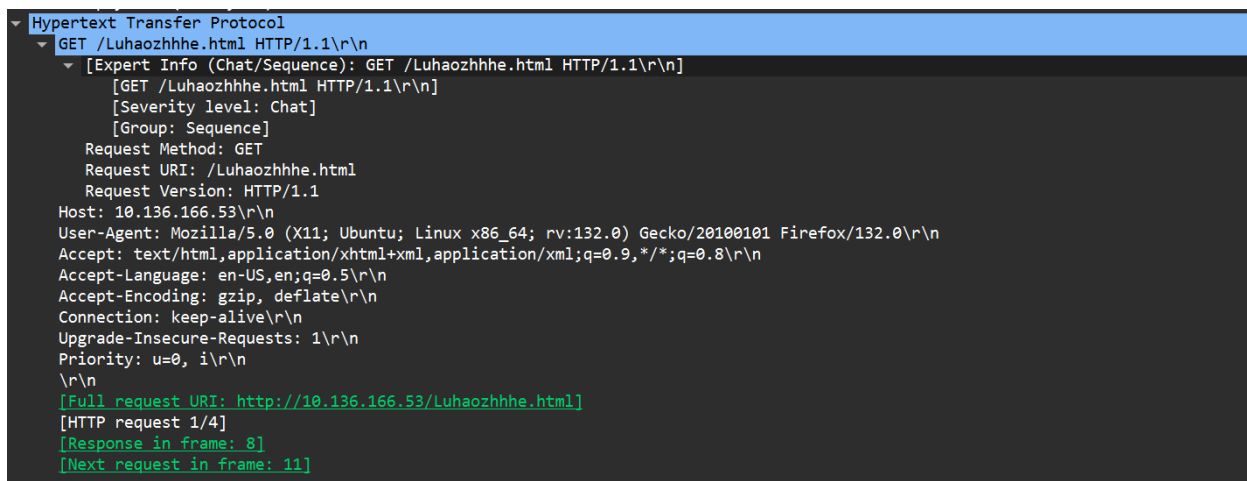


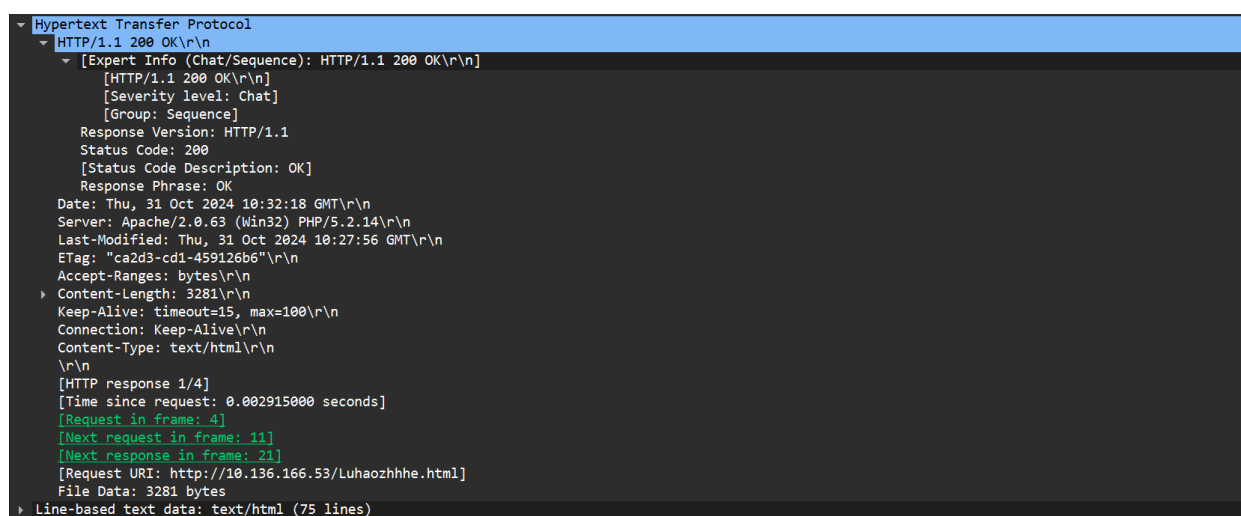
图 5.17: html 文件详细内容

首先，是我们请求行的内容。我们可以得到，GET 是我们的请求方法；/Luhaozhhe.html 是我们的请求 URL；HTTP/1.1 是我们的客户端与服务器的通信协议。

然后就来到我们的请求头部分。我们对其进行详细的分析：

- **Host:** 请求的服务器地址，为用户本机 IP 地址，也就是我们的服务器端的地址；
- **User-Agent:** Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:132.0) Gecko/20100101 Firefox/132.0, 表示客户端使用的火狐浏览器信息；
- **Accept:** text/html,application/xhtml+xml,application/xml, 指定客户端能够接收的内容类型；
- **Accept-Language:** zh-CN, 指定客户端接收的语言类型；
- **Accept-Encoding:** gzip, 指定客户端可接受的内容编码；
- **Connection:** keep-Alive, 表示这是一次长连接，可以处理多个请求。

然后，我们查看对应的响应头的部分。



```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
    [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
    [HTTP/1.1 200 OK\r\n]
    [Severity level: Chat]
    [Group: Sequence]
    Response Version: HTTP/1.1
    Status Code: 200
    [Status Code Description: OK]
    Response Phrase: OK
    Date: Thu, 31 Oct 2024 10:32:18 GMT\r\n
    Server: Apache/2.0.63 (Win32) PHP/5.2.14\r\n
    Last-Modified: Thu, 31 Oct 2024 10:27:56 GMT\r\n
    ETag: "ca2d3-cd1-459126b6"\r\n
    Accept-Ranges: bytes\r\n
    Content-Length: 3281\r\n
    Keep-Alive: timeout=15, max=100\r\n
    Connection: Keep-Alive\r\n
    Content-Type: text/html\r\n
    \r\n
    [HTTP response 1/4]
    [Time since request: 0.002915000 seconds]
    [Request in frame: 4]
    [Next request in frame: 11]
    [Next response in frame: 21]
    [Request URI: http://10.136.166.53/Luhaozhhe.html]
    File Data: 3281 bytes
  Line-based text data: text/html (75 lines)
```

图 5.18: html 文件响应头内容

1. 响应行:

- **Response Version:** 响应协议，为 HTTP/1.1;
- **Status Code:** 状态码，通常为 200，还有 3xx（重定向），4xx（客户端错误），5xx（服务器错误）；
- **Response Phrase:** OK，响应的描述。

2. 响应头:

- **Date:** Thu, 31 Oct 2024 10:27:56 GMT，为响应的时间；
- **Server:** 表示服务器的软件和版本等信息，这里为 Apache 和 PHP；
- **Content-Type:** text-html，表示内容的类型；

5.5.3 四次挥手

我们捕获到的结果中的最后四个为我们的四次挥手。

61	15.068107	192.168.188.131	10.136.166.53	TCP	54	60084 → 80	[FIN, ACK] Seq=1530 Ack=38584 Win=62780 Len=0
62	15.068338	10.136.166.53	192.168.188.131	TCP	54	80 → 60084	[ACK] Seq=38584 Ack=1531 Win=64239 Len=0
63	15.068658	10.136.166.53	192.168.188.131	TCP	54	80 → 60084	[FIN, PSH, ACK] Seq=38584 Ack=1531 Win=64239 Len=0
64	15.069083	192.168.188.131	10.136.166.53	TCP	54	60084 → 80	[ACK] Seq=1531 Ack=38585 Win=62780 Len=0

图 5.19: 四次挥手情况

下面我们分别进行详细的分析和解释。

首先是第一次挥手。

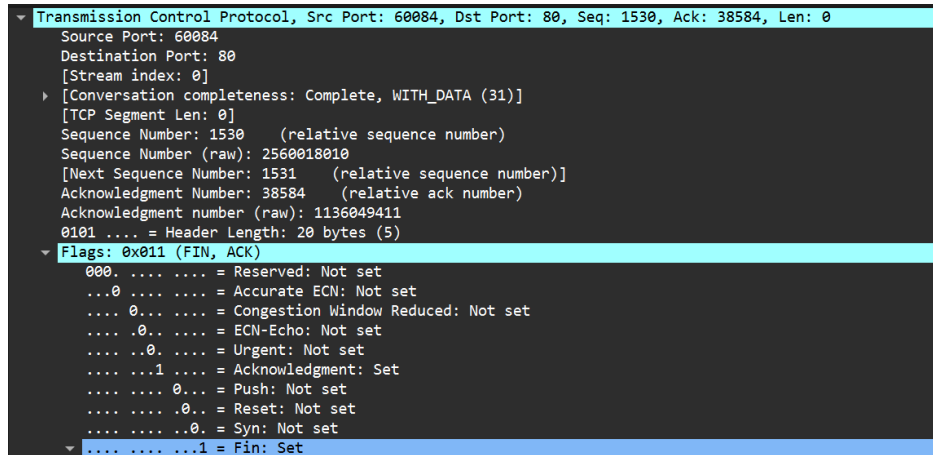


图 5.20: 第一次挥手

1. 这是由客户端发送给服务端的包，源端口和目的端口分别为 60084 和 80；
2. FIN 和 ACK 均被置位，FIN 用于表示主动关闭连接，通常只有 FIN 被置位时 ACK 也被置位，可能是由于已有数据的确认。如果在发送 FIN 标志位的同时还有之前接收到的数据需要被确认，那么 ACK 标志位也会被置位，且 ACK 标志位的设置表示该报文也携带着对之前接收到数据的确认；
3. 其确认号和序列号均与上一个 ACK 包相同；
4. 至此，客户端进入 FIN_WAIT1 状态。

然后是第二次挥手。

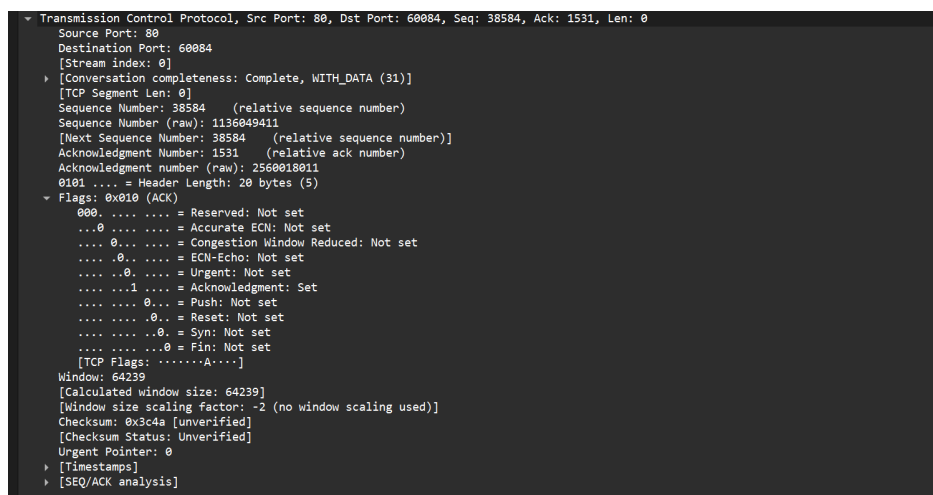


图 5.21: 第二次挥手

1. 这是由服务端发送给客户端的包，源端口为 80，目的端口为 60084；
2. ACK 被置位，用于表明服务端已经收到了上一个 FIN 包并作为应答。
3. 序列号与上一个 ACK 包相同，确认号为上一个序列号加 1，即 $1530+1 = 1531$ 。
4. 至此，服务端进入 CLOSE_WAIT 状态，客户端收到该包后进入 FIN_WAIT2 状态。

接着是第三次挥手。

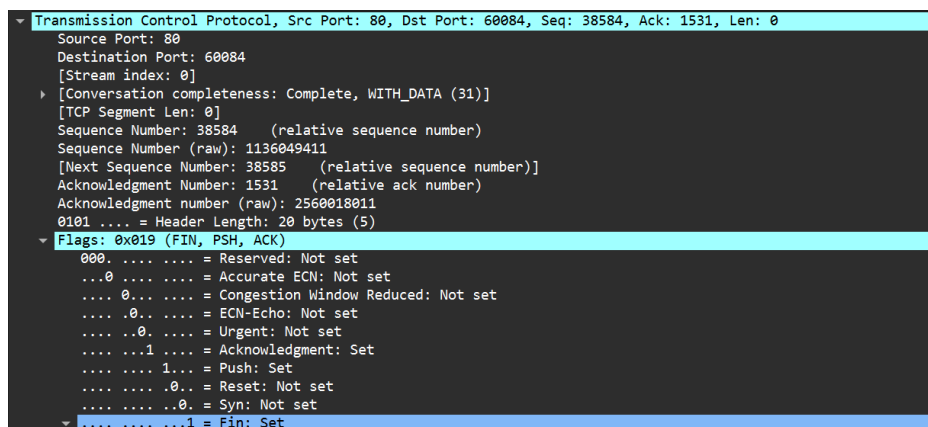


图 5.22: 第三次挥手

1. 这是由服务端发送给客户端的包，源端口和目的端口分别为 80 和 60084；
2. FIN 和 ACK 均被置位，即服务端通知客户端将断开连接；
3. 至此，服务端进入 LAST_ACK 状态；

最后是第四次挥手。

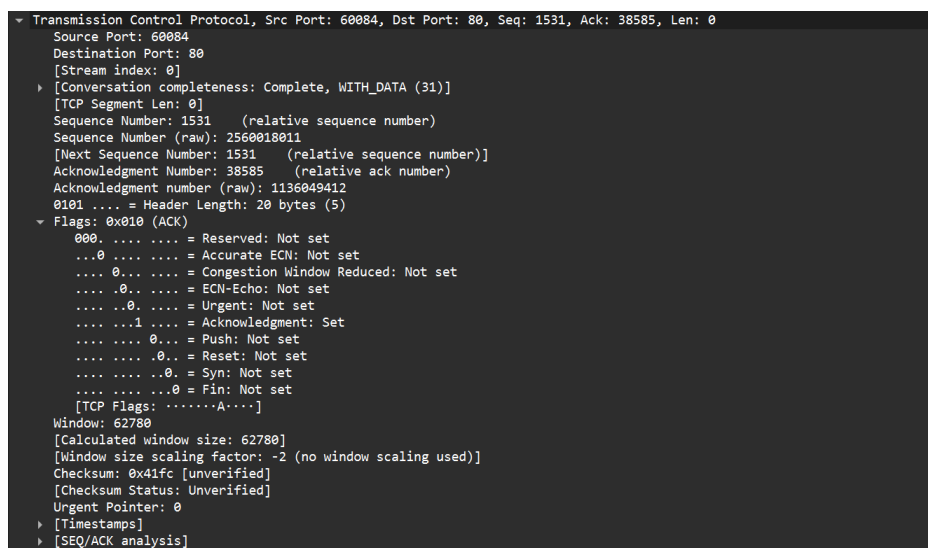


图 5.23: 第四次挥手

1. 这是由客户端发送给服务端的包，源端口和目的端口分别为 60084 和 80；

2. 这是最后一个 ACK 包，服务器接收后将关闭，序列号和确认号和前面是一样的，就不再说明了；
3. 至此，客户端进入 TIME_WAIT 状态，接下来将等待 2MSL，以防该 ACK 包发送失败。

这样，我们就分析完了整个抓到的包的具体流程和内容!!!

6 一些思考

1. 为什么是三次握手而不是两次？

- **确认可靠性：**三次握手能够确保双方都能接收到对方的连接请求和确认。两次握手可能导致连接不可靠，例如，如果客户端只发送一次 SYN 而没有收到 ACK，就无法确认服务器的状态；
- **避免错误连接：**通过三次握手，确保双方都是为了同一连接而进行通信，避免了错误连接的问题。比如，如果只用两次握手，可能导致重复的 SYN 报文被误解为新的连接请求；
- **保证同步：**三次握手确保了客户端和服务器在发送和接收信息时的状态是同步的，这对于 TCP 协议的可靠性是至关重要的。

2. 为什么是四次挥手而不是三次？

- **确保双方都能关闭连接：**四次挥手确保了双方都能独立地发送 FIN 报文段，表示没有更多数据要发送。这对于确保连接的正确关闭是至关重要的；
- **处理未完成的数据传输：**在服务器发送 FIN 之前，它可能仍有数据需要发送给客户端。通过这种方式，服务器可以在确认客户端关闭连接后，再关闭自己的连接；
- **避免数据丢失：**如果只用三次挥手，可能会导致一方在未确认的情况下关闭连接，从而丢失尚未传输完的数据。

3. 为什么第四次挥手后，主动方需等待 2MSL？

- **确保最后的 ACK 到达：**主动方在发送 ACK 后需要确保对方（被动方）能够收到这一报文。如果被动方没有收到 ACK，可能会重发 FIN。通过等待 2MSL，主动方可以确保有足够的时间来接收可能的重传，避免连接状态的不一致。
- **避免延迟的重复报文影响：**网络中可能存在延迟或重传的旧报文。如果主动方立即关闭连接，可能会导致这些旧报文被错误地解读为新的连接请求。等待 2MSL 可以确保这些旧报文在网络中消失，从而避免错误的连接建立。
- **保障资源的释放：**在 TIME_WAIT 状态下，主动方不会立即释放资源，这有助于保持 TCP 连接的完整性，并确保所有相关的报文都能被处理，防止出现潜在的资源冲突。
- **实现可靠性：**TCP 协议的设计宗旨是提供可靠的数据传输。通过确保所有报文的正确到达和处理，2MSL 等待时间增强了 TCP 连接的可靠性。

4. PSH 标志位有什么作用？

在 TCP 协议中，PSH (Push) 标志位的作用是告知接收方尽快将接收到的数据推送到应用层，而不需要等待缓冲区满或达到某个特定的条件。

- **立即处理数据：**当发送方设置了 PSH 标志位时，接收方的 TCP 栈会立即将接收到的数据传递给上层应用，而不是将数据存储在缓冲区中。这对于需要实时处理的数据流非常重要，比如交互式应用（如 SSH、Telnet 等）；
- **减少延迟：**通过设置 PSH 标志位，发送方可以减少数据传输的延迟，确保数据尽快到达应用层，提升用户体验；
- **适应交互式通信：**在许多交互式应用中，用户希望尽快看到反馈。PSH 标志位确保了这种需求的满足，使得应用能够迅速响应用户的操作；
- **控制数据流：**在某些情况下，即使接收方的缓冲区尚未满，发送方也可能希望将数据推送出去，以保持数据流的连贯性。PSH 标志位可以实现这种控制。

5. 捕获到的 keep-alive 是什么意思？

在 TCP 协议中，Keep-Alive 是一种机制，用于检测连接的活跃性和保持连接的状态。捕获到的 Keep-Alive 报文通常意味着发送方正在定期发送小的数据包，以确认连接仍然有效并且对方仍在响应。

在长时间没有数据传输的情况下，Keep-Alive 可以防止连接由于空闲而被网络设备（如防火墙）关闭。

当在网络抓包工具中捕获到 Keep-Alive 报文时，通常表示：

- 连接仍然保持活跃状态；
- 发送方在积极检测接收方的可用性；
- 网络连接可能经过了长时间的空闲，但仍希望维持连接以便于后续的数据传输。

6. 304 状态码是什么？

在抓包时有时会遇到 304 状态码，该状态码的含义是目前请求的信息与之前请求的内容相比没有改动，此时客户端从缓存读取即可，无须由服务器端再发送。

7 遇到的问题 & 解决方案

本次实验完成过程中，出现了很多意料之外的问题。下面由于篇幅原理，我就简单做一下说明。

首先就是我们的**音频大小问题**了！这个问题困扰了我好久。在刚开始，由于音频的时长和容量大小都过大，导致我们无法完成我们的抓包流程，最后居然只有三次挥手。而且会出现一堆红色的包，所以我上网进行了查询，发现是我们所存放的音频大小过大，导致我们抓包失败，后续对文件的大小进行了调整，就可以正常的抓包了！！

后续对我们抓包失败的那些情况进行了分析，发现：

- **TCP Window Full：**这通常表示接收方的 TCP 窗口已满，无法接收更多数据，因此发送方必须暂停发送数据，等待接收方扩大窗口后继续发送。这可能是由于网络拥塞、接收方资源限制或其他问题导致的；
- **TCP ZeroWindow：**这表示接收方的 TCP 窗口已经缩小到零，无法接收任何数据。这可能是由于接收方不愿意接收数据，或者由于资源限制或其他问题导致的。

然后，还有一个问题就是，**关于浏览器缓存的问题**。刚开始我们在浏览器中刷新页面，这时候，其实浏览器就已经存储了该文件的内容，之后进行访问时，只需要进行缓存的读取即可。而对应的，在我们的抓包界面，只会出现 304 Not Modified 的抓包情况。后续进行搜索，发现该状态其实也是正确的，只是因为该文件已经被浏览器存放在缓存中。我们如果不想看到这个状态，只需要在每次抓包前，都将浏览器中的缓存清空即可！

还有很多遇到的问题，此处就不再赘述了。

8 心得体会

通过本次实验，学习到了 html 前端界面的设计方式；学习到了利用 WireShark 进行抓包；分析了抓包的结果，并通过分析熟悉了通信的流程，包括三次握手、四次挥手等等。

当然，在实验中也遇到了很多问题。其中包括但不限于：对于音频大小过大而导致的窗口存满；对于未刷新浏览器缓存导致完成不了整个完整流程的抓包等等。当然，在一番尝试和搜索后，我都顺利地解决了这些问题。

对于一些报文中出现的位置，我还不是特别熟练，所以希望在接下来的课程学习中，可以将该部分的知识点学懂学精。