

南开大学

计算机网络课程实验报告

实验二：WireShark抓包



学院：网络空间安全学院

专业：信息安全

学号：2113997

姓名：齐明杰

班级：信安2班

1 实验要求

- 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。
- 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。

2 实验环境

2.1 Web服务器

本次实验我在本机运行Web服务器, 我采用Django框架配置Web服务器, 本机配置:

- **操作系统**: Microsoft Windows 10 专业版 (x64)
- **Wireshark版本**: Version 4.0.10 (v4.0.10-0)
- **Python版本**: 3.9.13

2.2 客户端

本次实验我使用Vmware虚拟机来作为客户端, 完成打开网页的操作。

- **虚拟环境**: VMware Workstation 17 Pro (17.0.2)
- **虚拟机**: Ubuntu 22.04
- **浏览器**: Firefox火狐浏览器

一个思考: 为什么不是直接使用本机的浏览器, 使用虚拟机作为客户端来访问?

主要考虑到以下几点优势:

1. 模拟实际网络环境:

- 通常情况下, Web服务器与客户端位于不同的物理或虚拟机上。通过使用虚拟机, 能够更真实地模拟实际的网络交互环境, 为进一步的网络实验提供了基础。

2. 清晰的网络交互:

- 利用虚拟机作为客户端, 可以在独立的网络接口上捕获数据包, 使得Wireshark抓包和分析过程更为清晰、简单, 减少了本地回环接口 (127.0.0.1) 可能带来的额外网络交互噪音。

3. 更真实的测试场景:

- 在独立的客户端系统上测试Web项目, 能够更好地理解项目在不同系统和网络配置下的行为, 有助于发现和解决潜在的兼容性问题。

2.3 IP地址

- 服务端: 192.168.148.1, 端口: 8000
- 客户端: 192.168.148.131

3 实验原理

3.1 Django Web服务器框架

Django是一个由Python编写的开源Web框架，它的设计理念是“快速开发”和“干净、明确的设计”。以下是关于Django框架的几个关键点：

1. 快速开发：

- Django提供了一套完善的工具和库，以帮助开发人员快速完成从概念到生产的过程。它努力减少开发人员需要做的冗余工作，使得开发人员可以专注于编写与项目需求直接相关的代码。

2. 清晰、明确的设计：

- 通过采用模型-模板-视图（MTV）设计模式，Django帮助开发人员组织代码，保持项目的清晰和可维护。同时，它也为开发人员提供了一个清晰、直观的编程框架，使得代码易于理解和扩展。

3. 强大的社区支持：

- Django拥有活跃的社区和丰富的文档资源，这为开发人员提供了学习资源和解决项目中遇到问题的途径，同时也有助于加速项目的开发和推进。

3.2 Http协议

在本次实验采用的是Http1.1作为Web环境，不使用Https和Http1.0。

3.2.1 Http的特点

HTTP（超文本传输协议）是互联网上应用最为广泛的一种网络协议，它是一个基于请求-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，客户端发出请求后，服务器会返回相应的响应。每个HTTP请求操作包含一个方法/动词，如GET（获取资源）、POST（提交数据）、PUT（更新资源）和DELETE（删除资源）等，这些方法指明了请求的目的。

2. 状态无关性：

- HTTP是一个状态无关的协议，每个请求都是独立的，不依赖于之前或之后的请求。然而，现代的Web应用通常通过如Cookie和Session等技术来维护状态信息。

3. 可扩展性与灵活性:

- 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。

HTTP协议的这些特点为Web应用的开发提供了基础，使得数据的传输和交互变得简单、明确且可靠。

3.2.2 HTTP协议请求和响应消息的主要组成部分

3.2.2.1 请求:

1. 请求行 (Request Line):

- 请求行包括请求方法（例如GET、POST等）、请求URI和HTTP版本。例如: `GET /index.html HTTP/1.1`。

2. 请求头 (Request Headers):

- 请求头包括了描述请求的各种头字段，例如 `Host`、`User-Agent`、`Accept` 等。

3. 请求体 (Request Body):

- 请求体包含了实际的请求数据。通常在POST或PUT请求中使用。

3.2.2.2 响应:

1. 状态行 (Status Line):

- 状态行包括HTTP版本、状态码和状态文本。例如: `HTTP/1.1 200 OK`。

2. 响应头 (Response Headers):

- 响应头包括了描述响应的各种头字段，例如 `Content-Type`、`Content-Length`、`Set-Cookie` 等。

3. 响应体 (Response Body):

- 响应体包含了实际的响应数据，通常是HTML、图片或其他媒体内容。

3.2.2.3 公共头部字段（通用于请求和响应）:

1. 通用头 (General Headers):

- 通用头字段包括如 `Cache-Control`、`Connection` 等，适用于请求和响应消息。

2. 实体头 (Entity Headers):

- 实体头字段包括如 `Content-Encoding`、`Content-Length` 等，描述消息体的属性。

3.2.2.4 消息体 (Message Body):

- 消息体包含了实际的请求或响应数据。在请求中，它可能包括表单数据或文件上传；在响应中，它可能包括HTML页面、图片或视频等。

3.2.3 HTTP 1.0和HTTP 1.1的区别

1. 连接的使用:

- HTTP 1.0每次请求后会关闭连接，需要为每个请求建立新的连接。而HTTP 1.1支持**持久连接**，允许在一个连接上完成多个请求和响应，**减少了TCP连接的建立和关闭的开销**。

2. 分块传输编码:

- HTTP 1.1引入了**分块传输编码** (chunked transfer encoding)，允许服务器在不知道消息体长度的情况下开始传输响应。

3. 主机头的要求:

- HTTP 1.1要求请求包含 `"Host"` 头，以便在同一台物理服务器上托管多个域名。

4. 管道化:

- HTTP 1.1引入了管道化，允许客户端在收到前一个响应之前发送多个请求，从而减少了网络的往返延迟。

3.2.4 HTTP和HTTPS的区别

1. 安全性:

- HTTP是不安全的，而HTTPS（超文本传输安全协议）通过SSL/TLS协议提供了一个安全的通信渠道。HTTPS能确保数据在传输过程中的机密性、完整性和身份验证。

2. 端口号:

- 默认情况下，HTTP使用端口80，而HTTPS使用端口443。

3. URL显示:

- HTTPS的URL以“https://”开头，而HTTP的URL以“http://”开头。

3.3 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。

以下是TCP协议报文的主要组成部分:

1. 源端口号和目标端口号 (Source Port and Destination Port):

- 源端口号和目标端口号分别标识了发送方和接收方的端口，每个端口号占用16位。

2. 序列号 (Sequence Number):

- 序列号用于标识发送的数据字节的顺序，它是32位的字段。在连接建立时，序列号用于标识SYN报文和第一个数据字节的序号。

3. 确认号 (Acknowledgment Number):

- 确认号是一个32位的字段，用于确认接收到的数据。它表示的是接收方期望接收的下一个字节的序号。

4. 数据偏移 (Data Offset) 或 头部长度 (Header Length):

- 数据偏移是一个4位的字段，指明了TCP头部的长度，从而标识数据部分的开始位置。

5. 保留位 (Reserved):

- 保留位是一个6位的字段，为未来使用而保留，通常设置为0。

6. 控制位 (Control Flags):

- 控制位是一个9位的字段，包含了如SYN、ACK、FIN、RST、URG等标志位，用于控制TCP的不同功能。

7. 窗口大小 (Window Size):

- 窗口大小是一个16位的字段，指明了接收方可接受的字节的数量，用于流量控制。

8. 校验和 (Checksum):

- 校验和是一个16位的字段，用于检查头部和数据的错误。

9. 紧急指针 (Urgent Pointer):

- 紧急指针是一个16位的字段，当URG标志位设置时使用，指明紧急数据的结束位置。

10. 选项和填充 (Options and Padding):

- 选项字段用于指定TCP的额外选项，例如最大报文段长度（MSS）、时间戳和窗口扩大因子等。

11. 数据 (Data):

- 数据字段包含了实际的应用数据。它的长度可以从数据偏移字段计算得出。

3.4 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报文：**客户端选择一个初始序列号 x ，并发送一个SYN报文，其中SYN标志位设置为1，序列号设置为 x 。

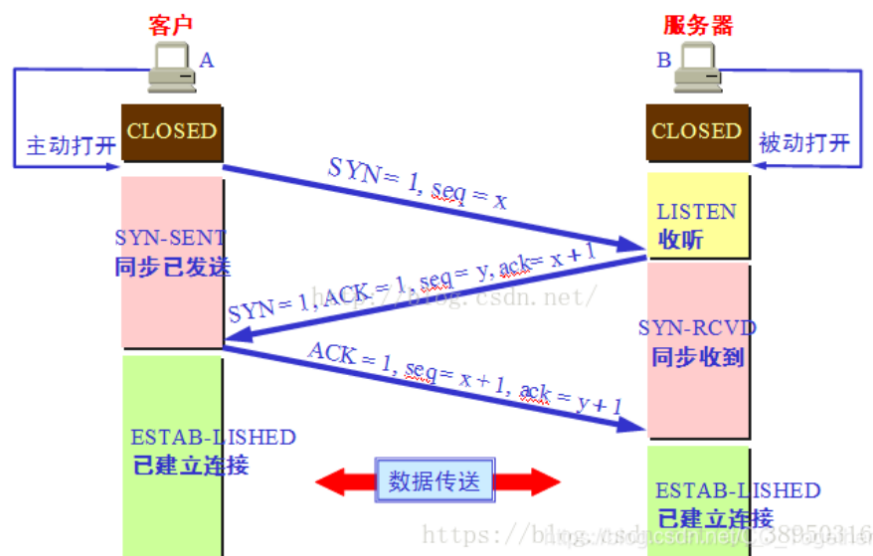
2. 第二次握手：

- **服务器发送SYN-ACK报文：**服务器收到客户端的SYN报文后，选择一个初始序列号 y ，并发送一个SYN-ACK报文，其中SYN标志位和ACK标志位都设置为1，序列号设置为 y ，确认号设置为 $x+1$ 。

3. 第三次握手：

- **客户端发送ACK报文：**客户端收到服务器的SYN-ACK报文后，发送一个ACK报文，其中ACK标志位设置为1，序列号设置为 $x+1$ ，确认号设置为 $y+1$ 。

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。



3.5 四次挥手

TCP四次挥手是断开TCP连接的过程，包括以下步骤：

TCP四次挥手的主要目的是为了确保双方都准备好断开连接，并且所有的数据都已经被成功传输。

1. 第一次挥手：

- **主动方发送FIN报文：**当主动方准备好关闭连接时，它发送一个FIN报文，其中FIN标志位设置为1，序列号设置为 u 。

2. 第二次挥手:

- **被动方发送ACK报文**: 被动方收到FIN报文后, 发送一个ACK报文, 其中ACK标志位设置为1, 序列号设置为 v , **确认号设置为 $u+1$** , 确认收到主动方的FIN报文。

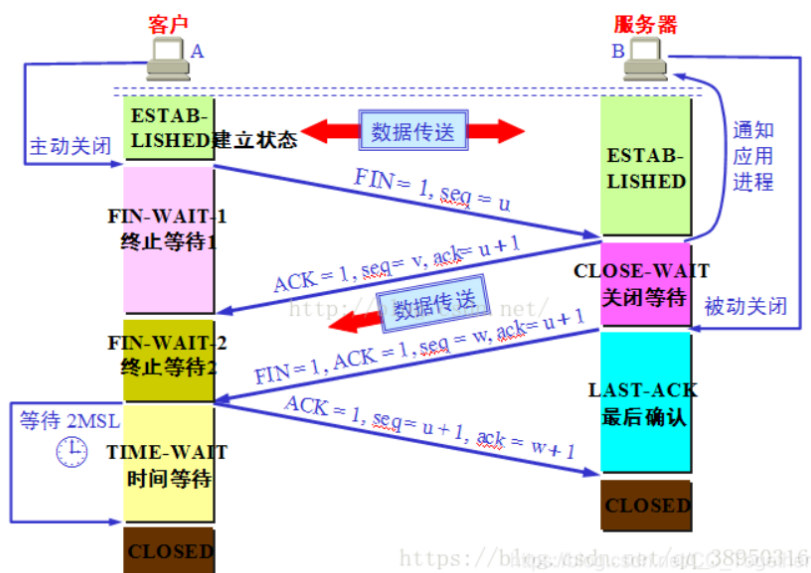
3. 第三次挥手:

- **被动方发送FIN-ACK报文**: 当被动方也准备好关闭连接时, 它发送另一个FIN报文, 其中FIN标志位和ACK标志位设置为1, 序列号设置为 w , **确认号设置为 $u+1$** 。

4. 第四次挥手:

- **主动方发送ACK报文**: 主动方收到被动的FIN报文后, 发送一个ACK报文, 其中ACK标志位设置为1, 序列号设置为 $u+1$, **确认号设置为 $w+1$** , 确认收到被动的FIN报文。

完成以上四个步骤后, TCP连接被成功断开。

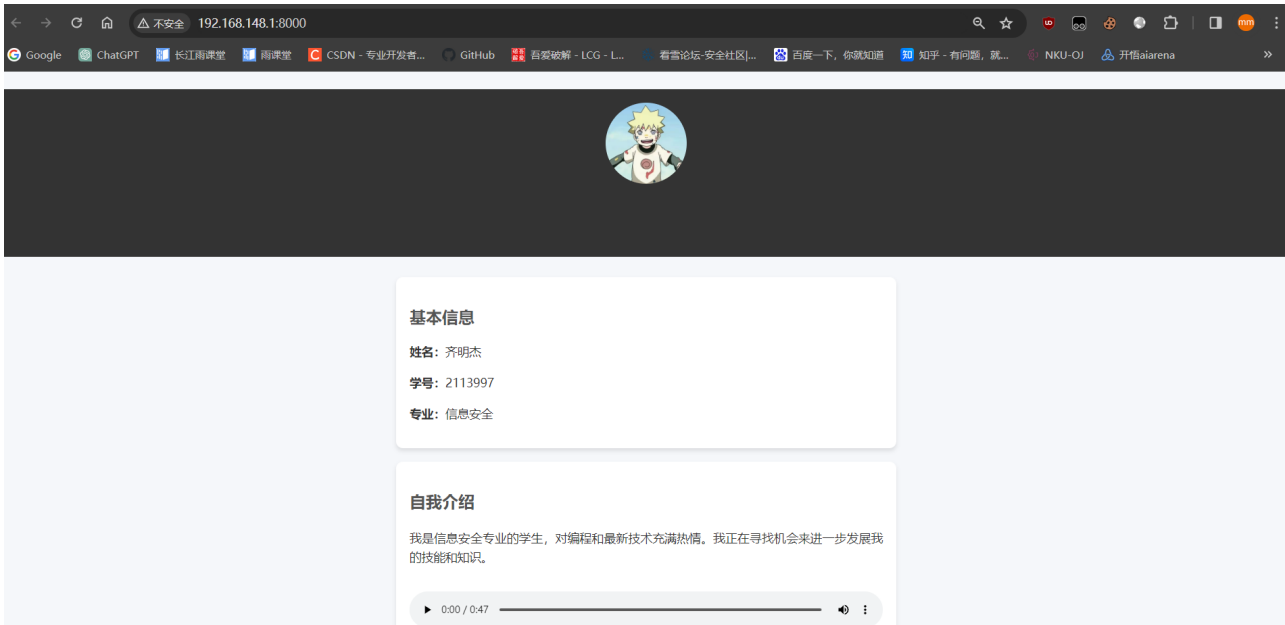


4 实验过程

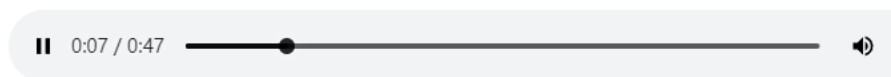
4.1 素材准备

4.1.1 HTML页面设计

Html页面设计追求简洁清晰, 同时配色合适即可, 页面预览如下:



点击音频播放按钮，可以正常播放音频：



代码使用了CSS渲染，部分截图如下：

```
3 <!DOCTYPE html>
4 <html lang="en">
5 <head>
6   <meta charset="UTF-8">
7   <title>个人信息</title>
8   <style>
9     body {
10       font-family: 'Nunito', sans-serif;
11       margin: 0;
12       padding: 0;
13       background: #f5f7fa;
14       color: #333;
15       line-height: 1.6;
16     }
17     .centered {
18       display: flex;
19       align-items: center;
20       justify-content: center;
21       min-height: 100vh;
22       flex-direction: column;
23     }
24     .card {
25       background: white;
26       border-radius: 10px;
27       padding: 20px;
```

4.1.2 Logo & 音乐

音频我选取了游戏《传送门2》的一个bgm，时长0:47。

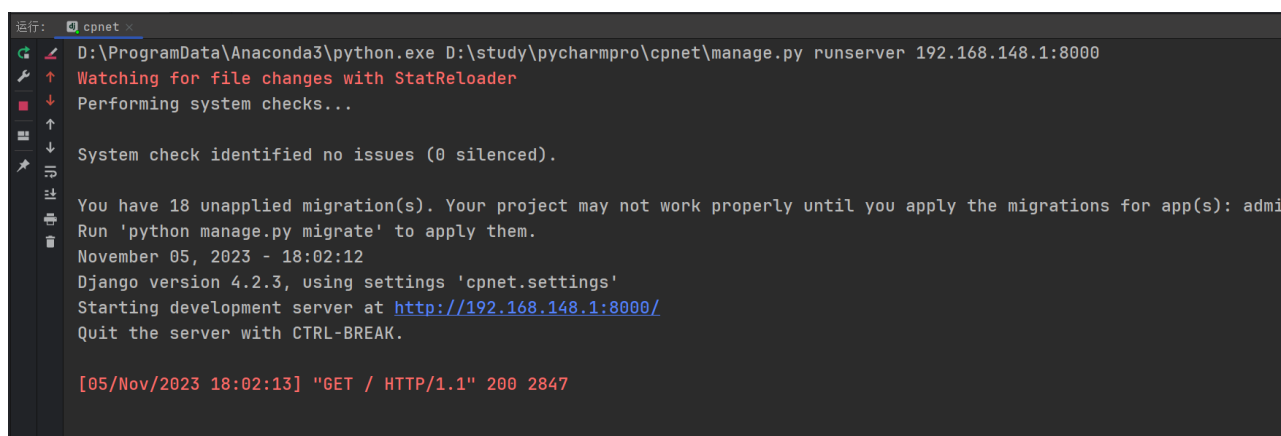
Logo是动漫《火影忍者》的一张图片，格式为jpg。

4.2 运行Web服务器

使用Python完成框架内容的编写，运行以下命令：

```
1 python.exe D:\study\pycharmpro\cpnet\manage.py runserver 192.168.148.1:8000
```

即可启动服务器于IP：192.168.148.1，端口：8000，如下所示：



```
运行: cpnet
D:\ProgramData\Anaconda3\python.exe D:\study\pycharmpro\cpnet\manage.py runserver 192.168.148.1:8000
Watching for file changes with StatReloader
Performing system checks...

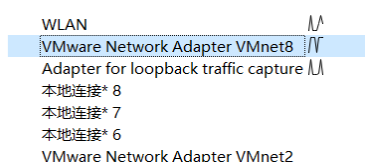
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin
Run 'python manage.py migrate' to apply them.
November 05, 2023 - 18:02:12
Django version 4.2.3, using settings 'cpnet.settings'
Starting development server at http://192.168.148.1:8000/
Quit the server with CTRL-BREAK.

[05/Nov/2023 18:02:13] "GET / HTTP/1.1" 200 2847
```

4.3 Wireshark启动

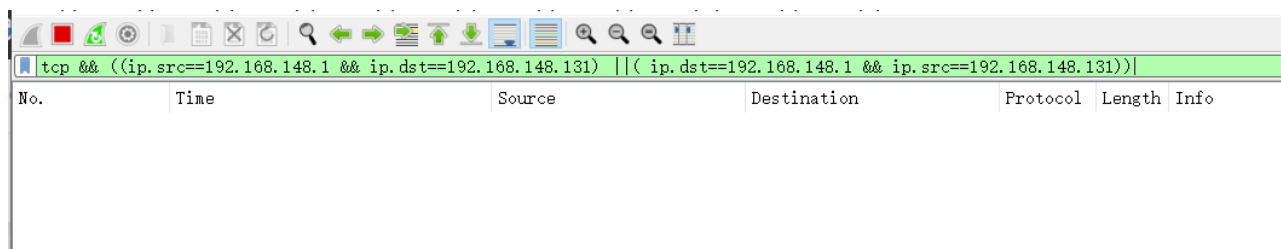
启动服务器后打开Wireshark，选择VMnet8：



然后在过滤器输入以下内容：

```
1 tcp && ((ip.src==192.168.148.1 && ip.dst==192.168.148.131) || (
  ip.dst==192.168.148.1 &&
2 ip.src==192.168.148.131))
```

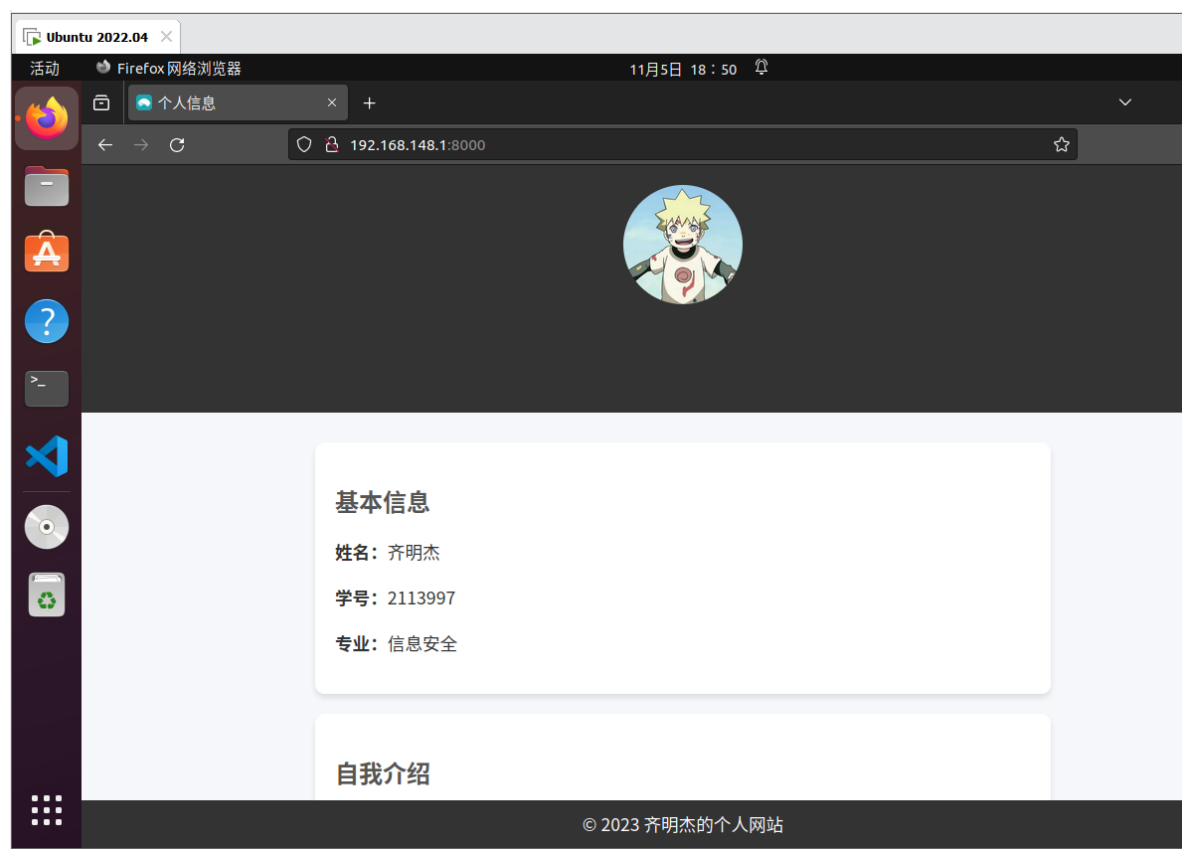
即只显示tcp协议，以及源IP和目标IP是本机和虚拟机(或虚拟机和本机)。



至此准备就绪，访问网页即可抓到TCP包了。

4.4 客户端访问网页

在虚拟机浏览器打开网址 `http://192.168.148.1:8000/` :



一切正常。

这样我们就捕获到了想要的包，包括TCP和HTTP协议，包括三次握手和四次挥手：

| | | | | | | | | | |
|----|------------|-----------------|-----------------|-----------------|------|------|-----------------------------------|------------|--|
| 1 | 2023-11-04 | 23:47:23.144671 | 192.168.148.131 | 192.168.148.1 | TCP | 74 | 50570 → 8000 | [SYN] | Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM |
| 2 | 2023-11-04 | 23:47:23.144933 | 192.168.148.1 | 192.168.148.131 | TCP | 74 | 8000 → 50570 | [SYN, ACK] | Seq=0 Ack=1 Win=65535 Len=0 MSS=146 |
| 3 | 2023-11-04 | 23:47:23.145094 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=1 Ack=1 Win=64256 Len=0 TSval=194604 |
| 4 | 2023-11-04 | 23:47:23.145200 | 192.168.148.131 | 192.168.148.1 | HTTP | 461 | GET / HTTP/1.1 | | |
| 5 | 2023-11-04 | 23:47:23.146784 | 192.168.148.1 | 192.168.148.131 | TCP | 83 | 8000 → 50570 | [PSH, ACK] | Seq=1 Ack=396 Win=131328 Len=17 TSV |
| 6 | 2023-11-04 | 23:47:23.146845 | 192.168.148.1 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [PSH, ACK] | Seq=18 Ack=396 Win=131328 Len=1448 |
| 7 | 2023-11-04 | 23:47:23.146845 | 192.168.148.1 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=1466 Ack=396 Win=131328 Len=1448 TSV |
| 8 | 2023-11-04 | 23:47:23.146845 | 192.168.148.1 | 192.168.148.131 | HTTP | 284 | HTTP/1.1 200 OK (text/html) | | |
| 9 | 2023-11-04 | 23:47:23.146918 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=396 Ack=18 Win=64256 Len=0 TSval=194 |
| 10 | 2023-11-04 | 23:47:23.146936 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=396 Ack=1466 Win=64000 Len=0 TSval=1 |
| 11 | 2023-11-04 | 23:47:23.146951 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=396 Ack=3132 Win=62336 Len=0 TSval=1 |
| 12 | 2023-11-04 | 23:47:23.190040 | 192.168.148.131 | 192.168.148.1 | HTTP | 427 | GET /static/img/logo.jpg HTTP/1.1 | | |
| 13 | 2023-11-04 | 23:47:23.191271 | 192.168.148.131 | 192.168.148.1 | TCP | 83 | 8000 → 50570 | [PSH, ACK] | Seq=3132 Ack=757 Win=130816 Len=17 |
| 14 | 2023-11-04 | 23:47:23.191334 | 192.168.148.131 | 192.168.148.1 | TCP | 1514 | 8000 → 50570 | [PSH, ACK] | Seq=3149 Ack=757 Win=130816 Len=144 |
| 15 | 2023-11-04 | 23:47:23.191334 | 192.168.148.131 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=4597 Ack=757 Win=130816 Len=1448 TSV |
| 16 | 2023-11-04 | 23:47:23.191334 | 192.168.148.131 | 192.168.148.131 | TCP | 1489 | 8000 → 50570 | [PSH, ACK] | Seq=6045 Ack=757 Win=130816 Len=142 |
| 17 | 2023-11-04 | 23:47:23.191366 | 192.168.148.131 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=7468 Ack=757 Win=130816 Len=1448 TSV |
| 18 | 2023-11-04 | 23:47:23.191366 | 192.168.148.131 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=8916 Ack=757 Win=130816 Len=1448 TSV |
| 19 | 2023-11-04 | 23:47:23.191366 | 192.168.148.131 | 192.168.148.131 | TCP | 1266 | 8000 → 50570 | [PSH, ACK] | Seq=10364 Ack=757 Win=130816 Len=12 |
| 20 | 2023-11-04 | 23:47:23.191396 | 192.168.148.131 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=11564 Ack=757 Win=130816 Len=1448 TS |
| 21 | 2023-11-04 | 23:47:23.191396 | 192.168.148.131 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=13012 Ack=757 Win=130816 Len=1448 TS |

| | | | | | | | | | | | | |
|------|------------|-----------------|-----------------|-----------------|------|------|----------------------|--------------|----------------|-------------|-------------|---------|
| 2106 | 2023-11-04 | 23:47:23.403509 | 192.168.148.1 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=2591655 | Ack=1569 | Win=1053696 | Len=144 |
| 2107 | 2023-11-04 | 23:47:23.403509 | 192.168.148.1 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=2593103 | Ack=1569 | Win=1053696 | Len=144 |
| 2108 | 2023-11-04 | 23:47:23.403509 | 192.168.148.1 | 192.168.148.131 | TCP | 1266 | 8000 → 50570 | [PSH, ACK] | Seq=2594551 | Ack=1569 | Win=1053696 | Len=144 |
| 2109 | 2023-11-04 | 23:47:23.403524 | 192.168.148.1 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=2595751 | Ack=1569 | Win=1053696 | Len=144 |
| 2110 | 2023-11-04 | 23:47:23.403524 | 192.168.148.1 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=2597199 | Ack=1569 | Win=1053696 | Len=144 |
| 2111 | 2023-11-04 | 23:47:23.403524 | 192.168.148.1 | 192.168.148.131 | TCP | 1266 | 8000 → 50570 | [PSH, ACK] | Seq=2598647 | Ack=1569 | Win=1053696 | Len=144 |
| 2112 | 2023-11-04 | 23:47:23.403546 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=1569 | Ack=2558887 | Win=2466688 | Len=0 |
| 2113 | 2023-11-04 | 23:47:23.403552 | 192.168.148.1 | 192.168.148.131 | TCP | 1514 | 8000 → 50570 | [ACK] | Seq=2599847 | Ack=1569 | Win=1053696 | Len=144 |
| 2114 | 2023-11-04 | 23:47:23.403552 | 192.168.148.1 | 192.168.148.131 | HTTP | 577 | HTTP/1.1 | 200 OK | (image/x-icon) | | | |
| 2115 | 2023-11-04 | 23:47:23.403553 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=1569 | Ack=2587559 | Win=2447360 | Len=0 |
| 2116 | 2023-11-04 | 23:47:23.403660 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=1569 | Ack=2591655 | Win=2464768 | Len=0 |
| 2117 | 2023-11-04 | 23:47:23.403667 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=1569 | Ack=2601806 | Win=2457088 | Len=0 |
| 2118 | 2023-11-04 | 23:47:33.405165 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | [TCP Keep-Alive] | 50570 → 8000 | [ACK] | Seq=1568 | Ack=2601806 | Win=0 |
| 2119 | 2023-11-04 | 23:47:33.405260 | 192.168.148.1 | 192.168.148.131 | TCP | 66 | [TCP Keep-Alive ACK] | 8000 → 50570 | [ACK] | Seq=2601806 | Ack=1568 | Win=0 |
| 2120 | 2023-11-04 | 23:47:43.658241 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | [TCP Keep-Alive] | 50570 → 8000 | [ACK] | Seq=1568 | Ack=2601806 | Win=0 |
| 2121 | 2023-11-04 | 23:47:43.658353 | 192.168.148.1 | 192.168.148.131 | TCP | 66 | [TCP Keep-Alive ACK] | 8000 → 50570 | [ACK] | Seq=2601806 | Ack=1568 | Win=0 |
| 2122 | 2023-11-04 | 23:47:44.041550 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [FIN, ACK] | Seq=1569 | Ack=2601806 | Win=2466688 | Len=0 |
| 2123 | 2023-11-04 | 23:47:44.042680 | 192.168.148.1 | 192.168.148.131 | TCP | 66 | 8000 → 50570 | [ACK] | Seq=2601806 | Ack=1570 | Win=1053696 | Len=0 |
| 2124 | 2023-11-04 | 23:47:44.042741 | 192.168.148.1 | 192.168.148.131 | TCP | 66 | 8000 → 50570 | [FIN, ACK] | Seq=2601806 | Ack=1570 | Win=1053696 | Len=0 |
| 2125 | 2023-11-04 | 23:47:44.042834 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=1570 | Ack=2601807 | Win=2466688 | Len=0 |

4.5 Wireshark抓包

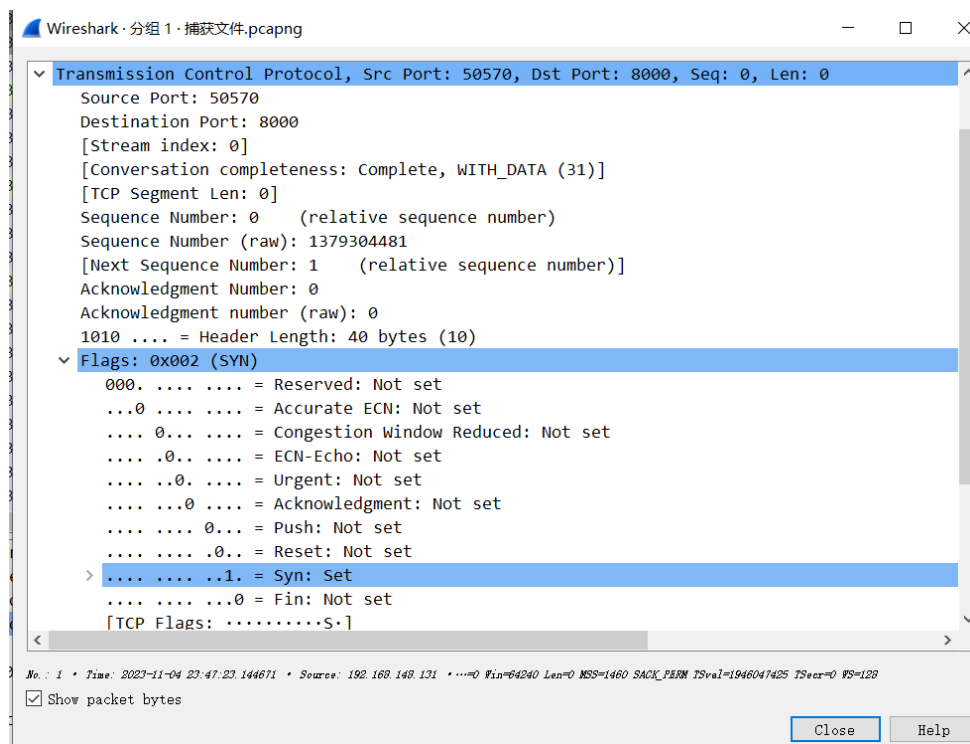
4.5.1 TCP三次握手

前三个包即为与三次握手相关的包：

| | | | | | | | | | | | | | |
|---|------------|-----------------|-----------------|-----------------|-----|----|--------------|------------|-------|-----------|-----------|----------|--------------|
| 1 | 2023-11-04 | 23:47:23.144671 | 192.168.148.131 | 192.168.148.1 | TCP | 74 | 50570 → 8000 | [SYN] | Seq=0 | Win=64240 | Len=0 | MSS=1460 | SACK_PERM |
| 2 | 2023-11-04 | 23:47:23.144933 | 192.168.148.1 | 192.168.148.131 | TCP | 74 | 8000 → 50570 | [SYN, ACK] | Seq=0 | Ack=1 | Win=65535 | Len=0 | MSS=1460 |
| 3 | 2023-11-04 | 23:47:23.145094 | 192.168.148.131 | 192.168.148.1 | TCP | 66 | 50570 → 8000 | [ACK] | Seq=1 | Ack=1 | Win=64256 | Len=0 | TSval=194604 |

下面对三个包展开具体的分析：

- 第一次握手：



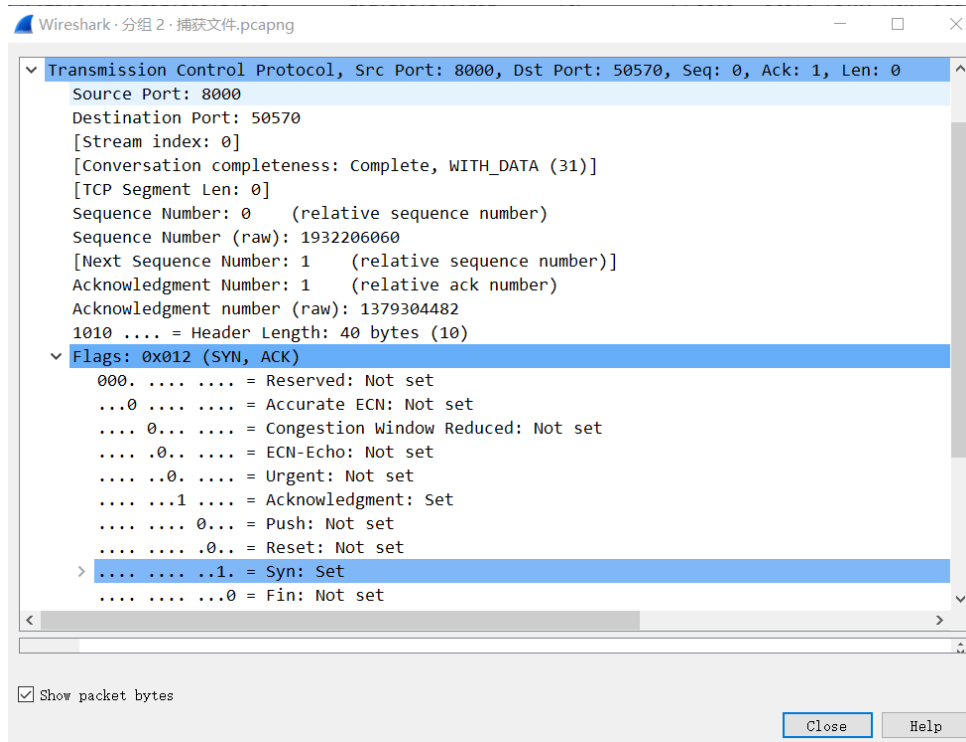
这是由客户端发送给服务端的包，前两行是源端口，目的端口，为50570和8000。

客户端随机生成了一个初始序列号，为 **1379304481**，当然这是绝对序列号，其相对序列号为0。

确认号(ack字段)的值我们可以不看，因为ACK标志位没有置位。

SYN被置位，表明这是一次建立连接请求，至此客户端进入 **SYN-SENT** 状态。

- 第二次握手:



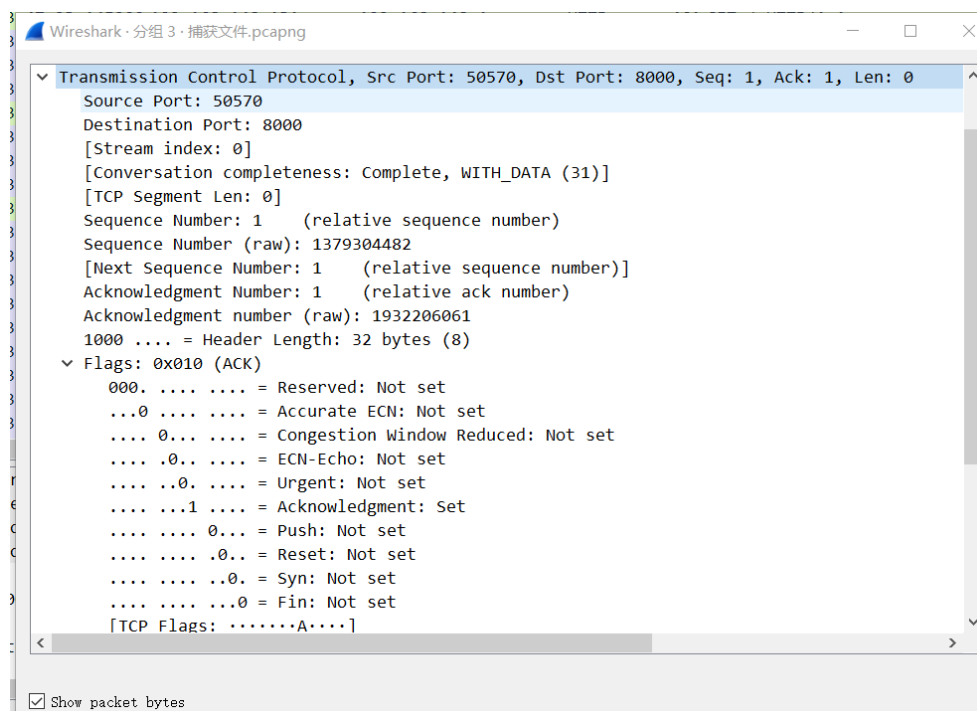
这是由服务端发送给客户端的包，源端口，目的端口为8000和50570。

服务端随机生成了一个初始序列号，为 1932206060，其相对序列号为0。

ACK被置位，确认号(ack字段)的绝对值为第一次握手的seq+1，即 $1379304481+1=1379304482$ ，相对值为1。

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。

- 第三次握手:



这是由客户端发送给服务端的包，源端口，目的端口为50570和8000。

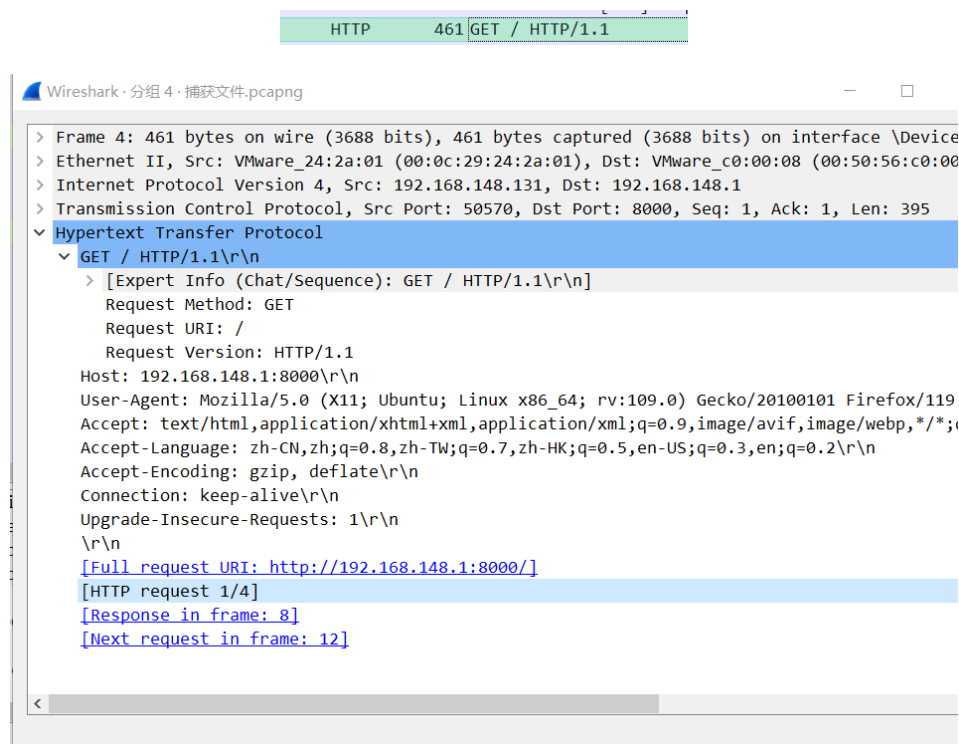
客户端的序列号为1(相对客户端上一次发包+1)。

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
 $1932206061+1=1932206062$ ，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。

4.5.2 Http请求

打开这个请求查看；



- 请求行

- 请求方法: GET
- 请求URI: /
- 请求协议: HTTP/1.1，客户端与服务器通信的协议

- 请求头

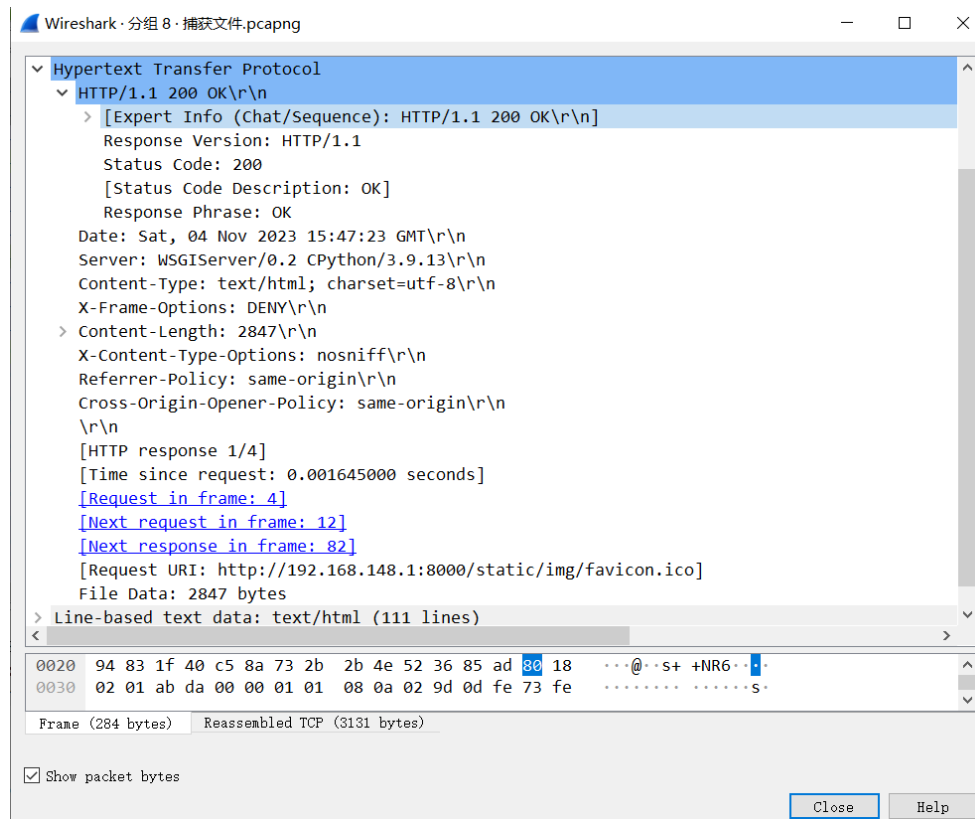
- Host: 请求的服务器地址，这是我本机的IP地址
- User-Agent: 这是火狐浏览器自动生成的用户代理，表示客户端使用的浏览器信息。

Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/119.0

- 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**: 指定客户端可接受的内容编码，此处为gzip。
- **Connection**: keep-Alive，表示这是一次长连接，可以处理多个请求。

接下来打开响应包查看：



• 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。

• 响应头：

- **Date**: Sat, 04 Nov 2023 15:47:23 GMT，为响应的时间
- **Server**: 表示服务器的软件和版本等信息，这里是WSGIServer和CPython3.9.13
- **Content-Type**: text-html，表示内容的类型。
- **X-Frame-Options**: 用于防止点击劫持攻击。
- **Content-Length**: 表示响应体的字节数。

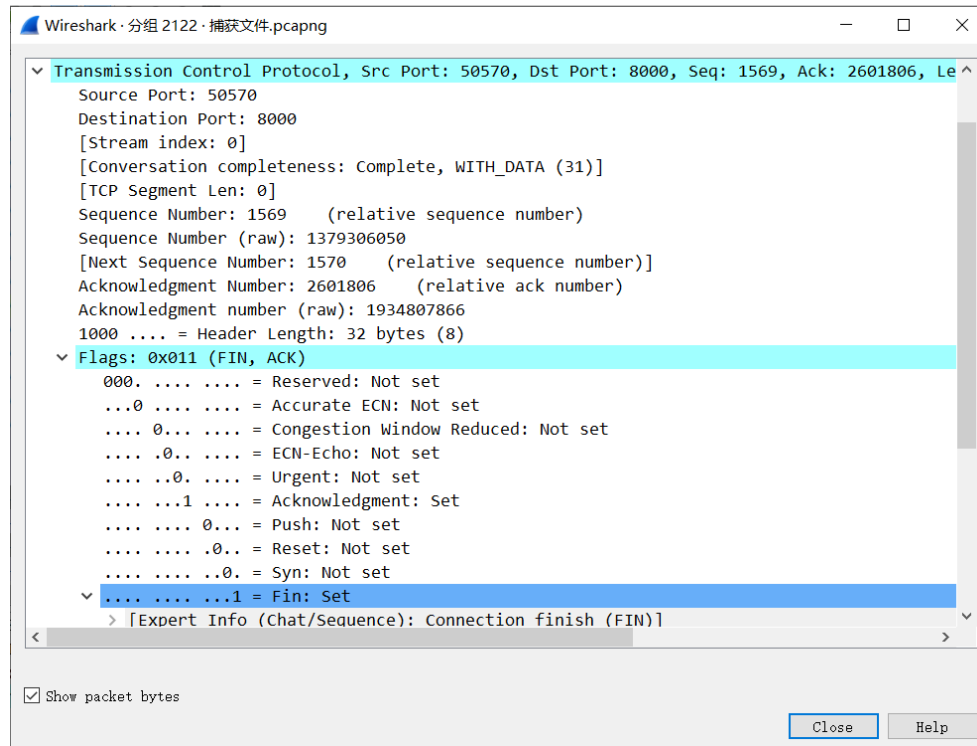
4.5.3 TCP四次挥手

后四个包即为与四次挥手相关的包：

| | | | | | | |
|------|------------|-----------------|-----------------|-----------------|-----|--|
| 2121 | 2023-11-04 | 23:47:43.658353 | 192.168.148.1 | 192.168.148.131 | TCP | 66 [TCP Keep-Alive ACK] 8000 → 50570 [ACK] Seq=2601806 Ack=156 |
| 2122 | 2023-11-04 | 23:47:44.041550 | 192.168.148.131 | 192.168.148.1 | TCP | 66 50570 → 8000 [FIN, ACK] Seq=1569 Ack=2601806 Win=2466688 Le |
| 2123 | 2023-11-04 | 23:47:44.042680 | 192.168.148.1 | 192.168.148.131 | TCP | 66 8000 → 50570 [ACK] Seq=2601806 Ack=1570 Win=1053696 Len=0 T |
| 2124 | 2023-11-04 | 23:47:44.042741 | 192.168.148.1 | 192.168.148.131 | TCP | 66 8000 → 50570 [FIN, ACK] Seq=2601806 Ack=1570 Win=1053696 Le |
| 2125 | 2023-11-04 | 23:47:44.042834 | 192.168.148.131 | 192.168.148.1 | TCP | 66 50570 → 8000 [ACK] Seq=1570 Ack=2601807 Win=2466688 Len=0 T |

下面对四个包展开具体的分析：

- 第一次挥手:



这是由客户端发送给服务端的包，源端口，目的端口为50570和8000.

FIN 和 **ACK** 均被置位，**FIN** 用于表示主动关闭连接，理论上只有**FIN**被置位，但**ACK**却也被置位，令人疑惑。

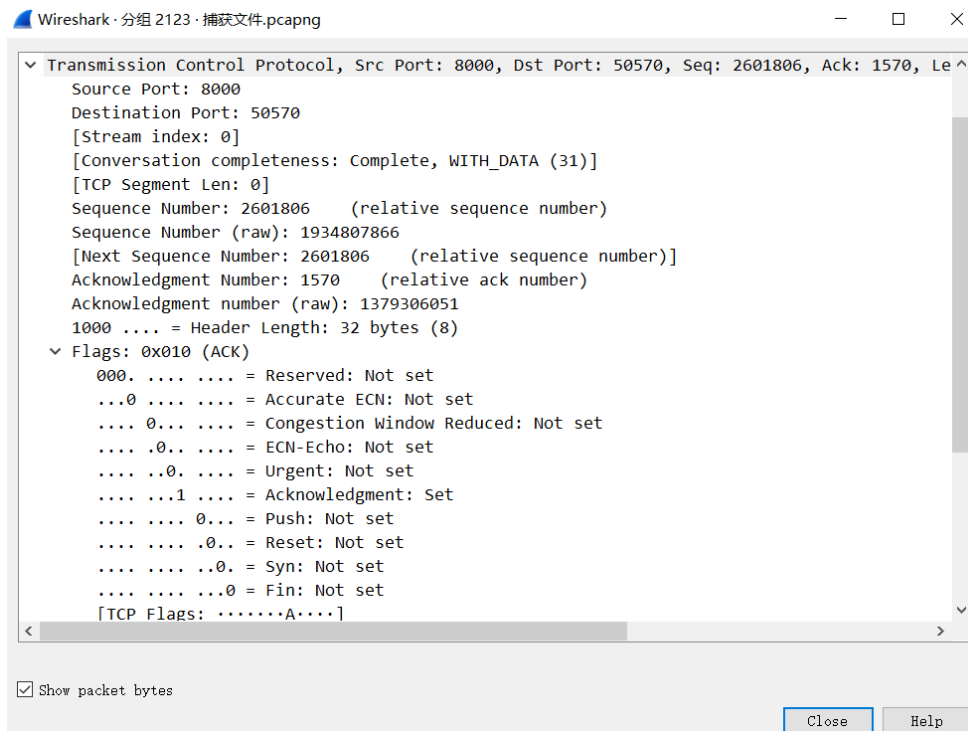
经查阅资料，可能是由于已有数据的确认：

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。**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。

其**确认号**和**序列号**，均与上一个ACK包相同。

至此，客户端进入 **FIN_WAIT1** 状态.

- 第二次挥手:



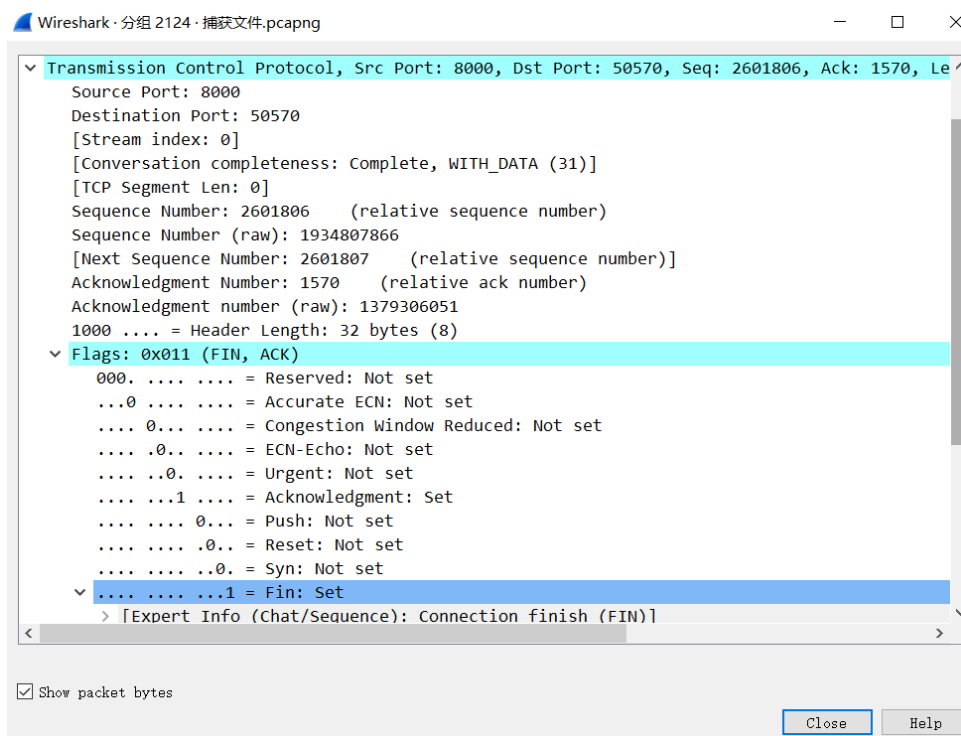
这是由服务端发送给客户端的包，源端口，目的端口为8000和50570。

ACK被置位，用于表明服务端已经收到了上一个FIN包，作为应答。

序列号与上一个ACK包相同，确认号为上一个Seq+1，即1569+1=1570

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

● 第三次挥手：

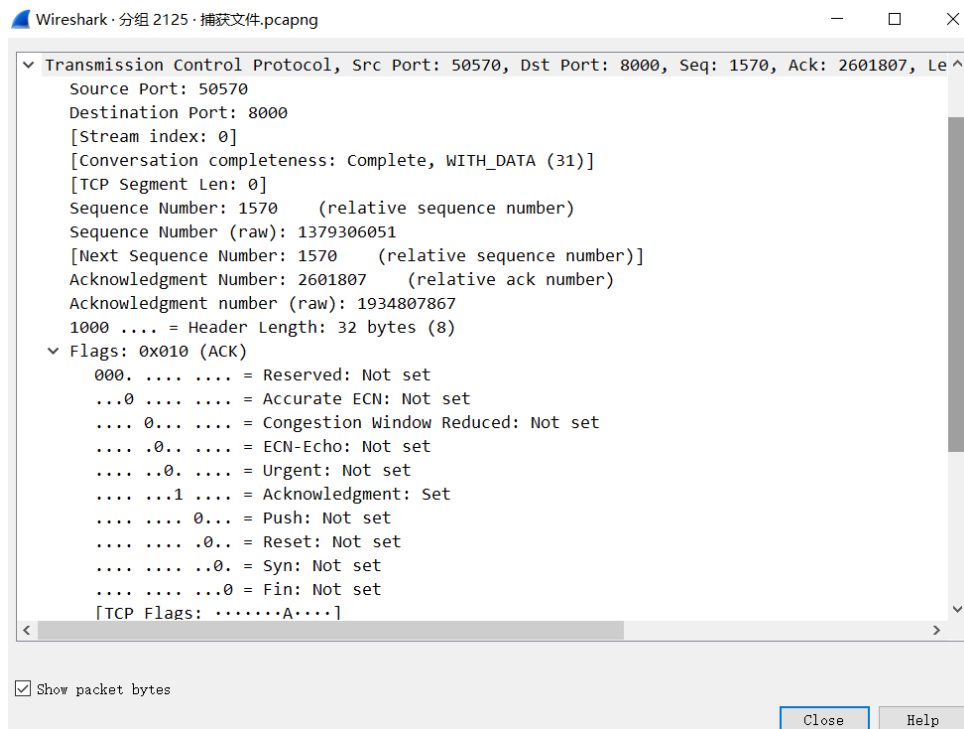


这是由服务端发送给客户端的包，源端口，目的端口为8000和50570。

`FIN` 和 `ACK` 均被置位，即服务端通知客户端将断开连接。

至此，服务端进入 `LAST_ACK` 状态。

- 第四次挥手：



这是由客户端发送给服务端的包，源端口，目的端口为50570和8000。

这是最后一个ACK包，服务器接收后将关闭，序列号和确认号同理，不再赘述。

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。

5 一些常见问题

5.1 PSH标志位有什么作用？

TCP协议中的 `PSH` (Push) 标志位是用于控制数据传送的。当 `PSH` 标志位被设置时，它指示接收端应该立即将这些数据推送给应用层，而不是等待缓冲区填满。这样可以确保数据能够尽快地被应用程序处理，而不是在TCP缓冲区中等待。

5.2 为什么是三次握手而不是两次？

TCP使用三次握手 (Three-Way Handshake) 而非两次握手来建立连接，主要是为了解决网络中的延迟和重复报文问题，确保连接的可靠性和数据传输的准确性。具体来说，有以下几点原因：

1. 确保双方通信能力：

- 三次握手可以确保双方都具备发送和接收数据的能力。在第一次和第二次握手中，服务器和客户端分别确认了对方的发送和接收能力；在第三次握手中，客户端再次确认了服

服务器的接收能力。

2. 防止过期的连接请求：

- 由于网络延迟，过期的连接请求可能会在网络中滞留并在稍后到达服务器。如果使用两次握手，服务器可能会基于过期的请求建立一个无效的连接。而三次握手可以通过客户端的最终确认来避免这种情况，因为过期的请求不会得到客户端的最终确认。

3. 初始化序列号：

- 在三次握手的过程中，双方交换了各自的初始序列号，这为后续的数据传输提供了序列号的同步基准，确保数据的有序传输。

5.3 为什么是四次挥手而不是三次？

TCP 使用四次挥手（Four-Way Handshake）来终止一个连接，主要是因为 TCP 是全双工协议，意味着数据可以在两个方向上独立传输。每个方向的关闭需要独立处理，因此需要四步操作来确保双方都完成了数据传输并且同意终止连接。具体来说，四次挥手的过程有以下几个原因和优势：

1. 全双工通信：

- TCP协议允许数据在两个方向上独立传输。因此，每个方向都需要单独关闭。一个方向的关闭不应影响另一个方向的数据传输，这就需要四次挥手来分别处理两个方向的连接终止。

2. 确保数据完整传输：

- 四次挥手确保了在关闭连接之前，双方都有足够的时间发送和接收剩余的数据。通过这种方式，它保证了在连接关闭之前，所有的数据都被正确和完整地传输。

3. 避免误关闭：

- 如果使用三次挥手，可能会出现一方还有数据需要发送，但另一方已经关闭连接的情况。四次挥手通过分开处理每个方向的连接关闭，避免了这种误关闭的情况。

4. 确认关闭：

- 四次挥手提供了足够的确认机制，确保双方都明白连接正在关闭，并且所有的数据都已经传输完毕。这种明确的确认机制增加了连接终止过程的可靠性。

5.4 为什么第四次挥手后，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"TIME-WAIT"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：

1. 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。

2. 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。

6 总结与感想

我在检查时对PSH标志位的作用不太清晰，但其他问题我均能流利地回答，经过查阅资料后，我对各项细节也更加熟悉，感谢助教的提问，这让我对TCP相关的内容有了更深入思考，而并非停留在浅层内容。通过这次实验，我对wireshark有了更熟练的使用，同时也巩固了Web内容的编写和框架使用。

PSH