计算机网络实验报告

Lab2 配置Web服务器，分析HTTP交互过程

网络空间安全学院 物联网工程专业 2212039 田晋宇

jassary08/Computer_Network (github.com)

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。

注：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。

服务器搭建

在本次实验中通过Node.js，使用 `Express` 和 `Parcel` 创建了一个简单的静态服务器，用于处理 HTTP 请求和响应。

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，它允许开发者在服务器端运行 JavaScript 代码。它的主要特点是单线程、非阻塞 I/O 操作和事件驱动架构，这使得它特别适合于构建高并发、实时响应的网络应用程序（如 Web 服务器、聊天应用、RESTful API 等）。

在终端中进入到项目文件的根目录，输入命令`npm start`，可以看到服务器成功建立。

```
PS F:\Desktop\Grade3\Computer_Network\lab2> npm start

> lab2@1.0.0 start
> node server.js

- Building...Server is running on http://127.0.0.1:3000
√  Built in 1.79s.
```

在浏览器中打开指定HTTP网址，即可打开个人主页：

服务器的建立主要通过**server.js**实现：

1. 引入模块

```
const express = require('express');
const path = require('path');
const Bundler = require('parcel-bundler');
```

- `express`：用于创建 HTTP 服务器的轻量级框架，简化了处理请求和响应的过程。
- `path`：Node.js 的内置模块，用于处理文件和目录路径。
- `parcel-bundler`：一个 Web 应用程序的打包工具，在这里作为中间件用于构建和提供静态文件。

2. 创建 Express 应用

```
const app = express();
```

使用 `express()` 创建一个应用实例 `app`，用于定义路由和中间件。

3. 设置 IP 地址和端口

```
const HOST = '127.0.0.1';
const PORT = 3000;
```

定义了服务器监听的 IP 地址和端口号。在本地运行时，使用 `127.0.0.1:3000` 访问服务器。

4. 配置静态文件目录

```
app.use(express.static(path.join(__dirname, 'public')));
```

- `express.static`：Express 提供的静态文件中间件，用于提供指定目录中的静态资源（如 HTML、CSS、JavaScript、图片等）。
- `path.join(__dirname, 'public')`：将当前目录下的 `public` 文件夹设为静态文件目录。
- 当浏览器请求一个资源时，Express 会先检查 `public` 文件夹中是否有该资源，如果有，则直接提供该文件。

5. 配置 Parcel 中间件

```
const bundler = new Bundler('public/index.html'); // 指定入口文件为
`public/index.html`
app.use(bundler.middleware());
```

- `new Bundler('public/index.html')`：初始化 Parcel，将 `public/index.html` 作为入口文件，Parcel 会将相关的资源（HTML、CSS、JavaScript 等）打包并提供给客户端。
- `app.use(bundler.middleware())`：将 Parcel 作为中间件添加到 Express 中。Parcel 会监听文件更改，并在浏览器请求时自动重新打包资源，适合开发环境。

6. 启动服务器

```
app.listen(PORT, HOST, () => {
    console.log(`Server is running on http://${HOST}:${PORT}`);
});
```

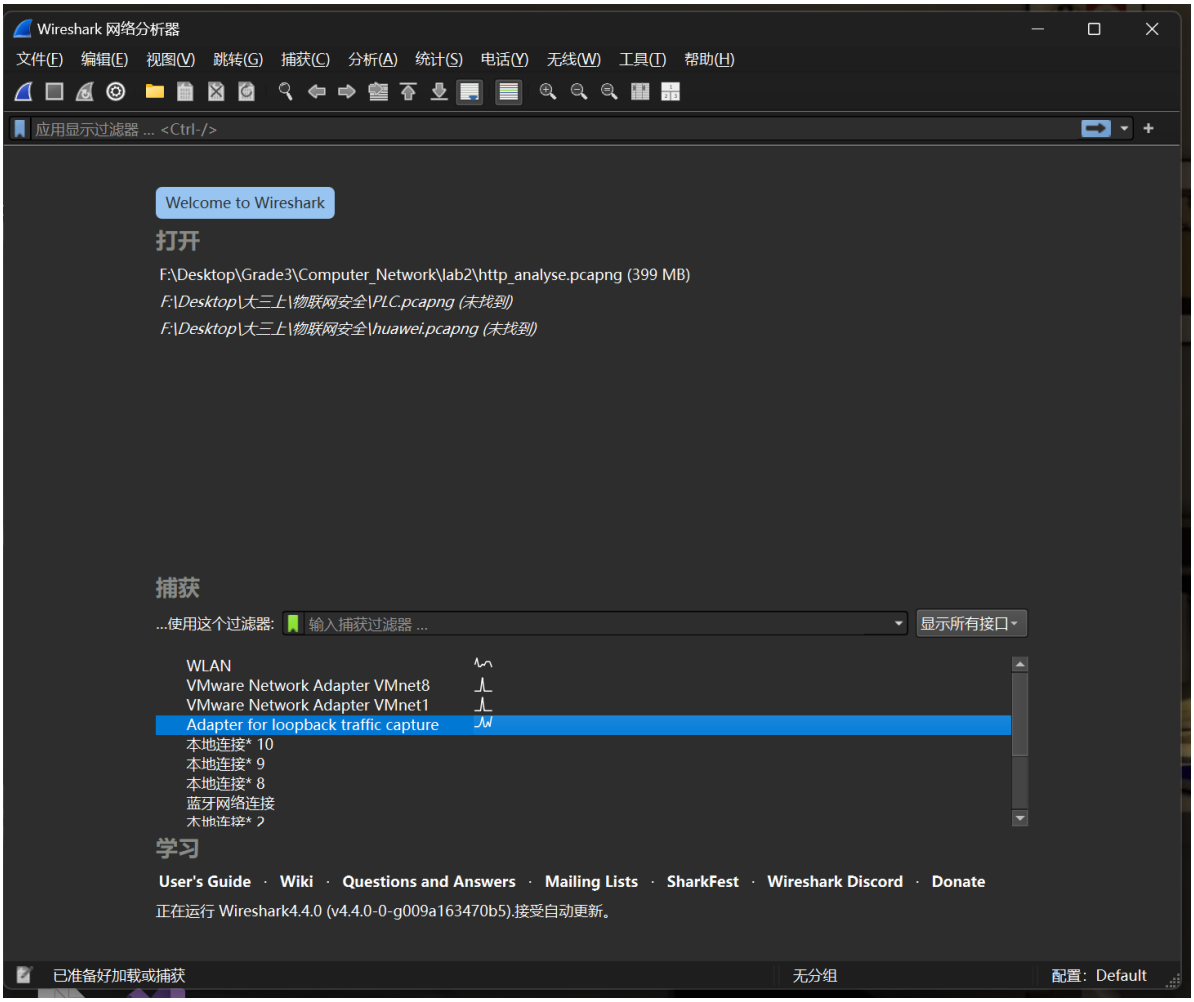
- `app.listen`：启动服务器，开始监听定义的 IP 地址和端口（即 `127.0.0.1:3000`）。
- `console.log` 输出一条消息，提示服务器已启动并显示访问地址。

抓包分析

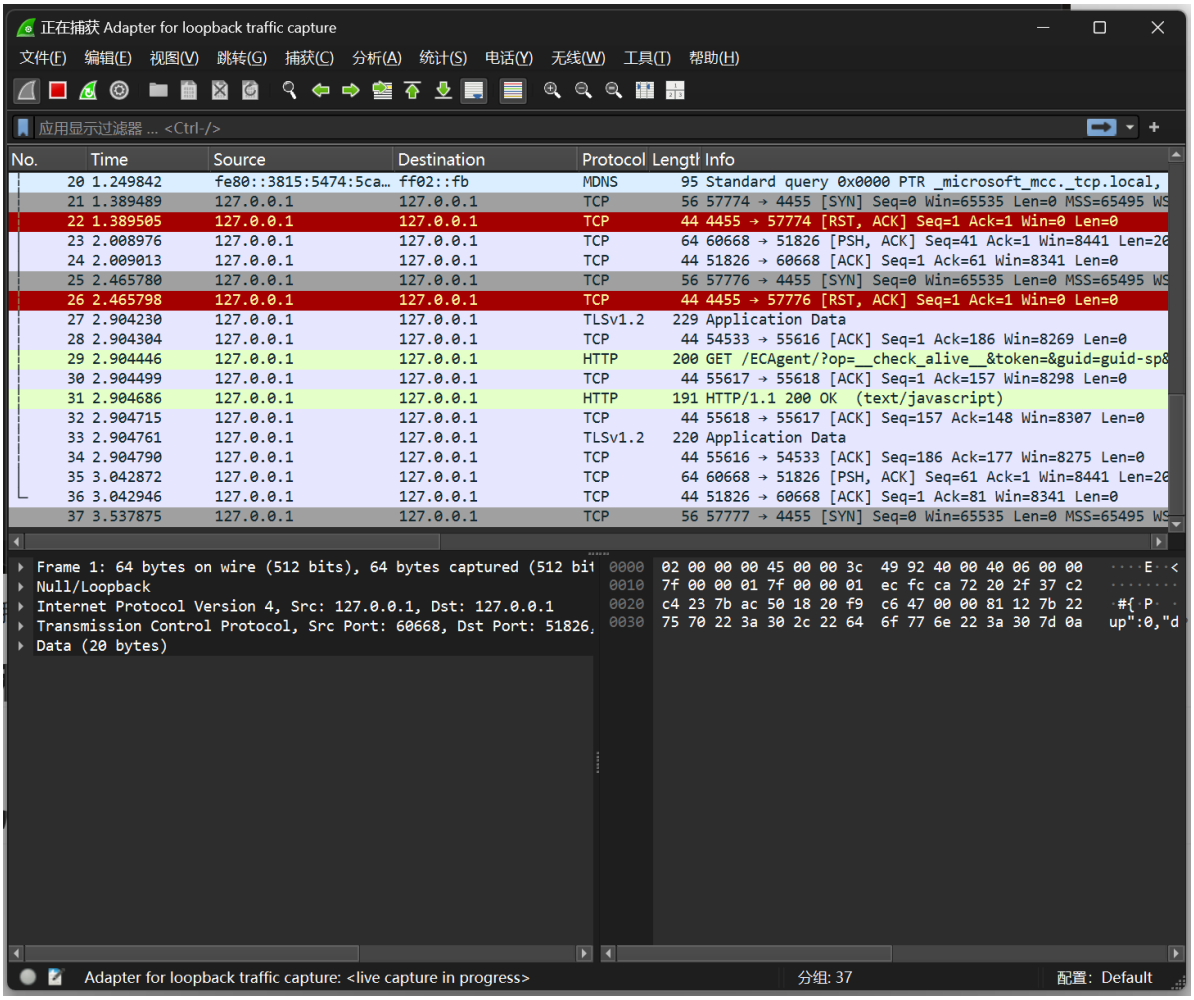
1. 数据捕获

本次实验我们使用wireshark对网络数据包进行抓包分析。Wireshark 是一款功能强大的网络协议分析工具，广泛应用于网络监控、故障排查、网络开发、教学等场景。它可以实时捕获网络数据包，并提供详细的协议解析，是网络工程师和安全专家常用的工具之一。

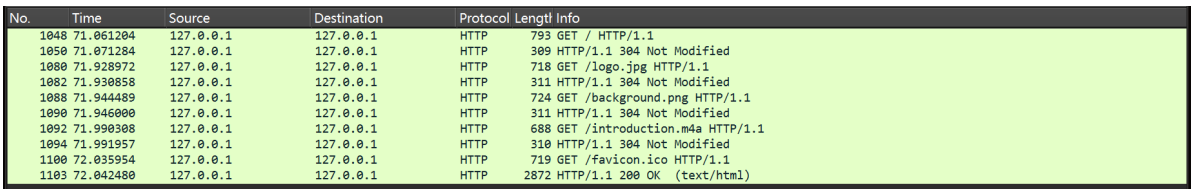
打开wireshark后，我们可以看到许多可以捕获的网络接口：



我们点击进入`Adapter for loopback traffic capture`,这个网络接口捕获计算机与自己进行网络通信的数据包。进入之后我们可以发现wireshark已经在源源不断的捕获数据包:



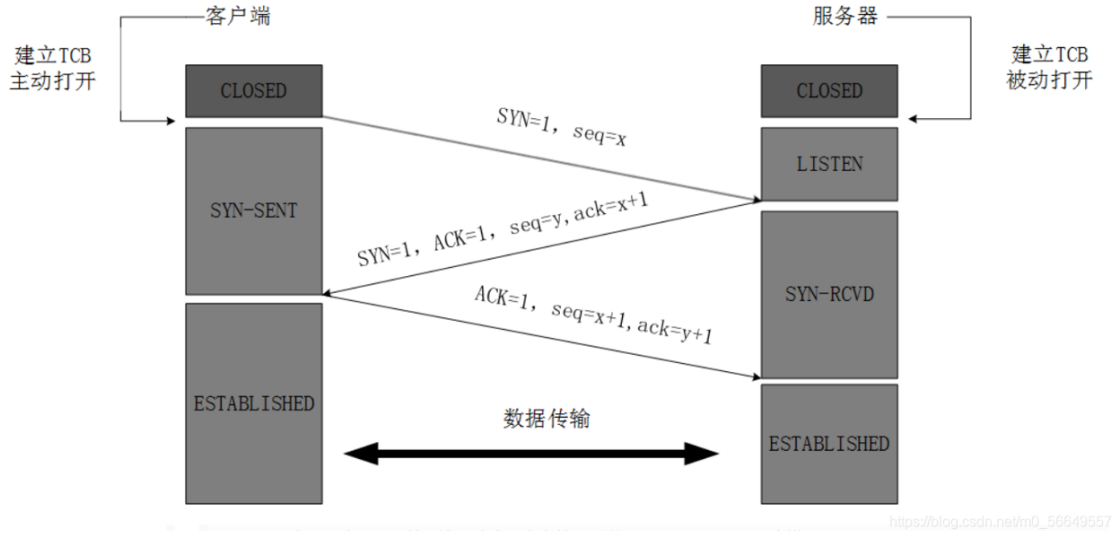
wireshark中有过滤器功能，通过过滤器功能我们捕获到我们想要的数据包。我们将过滤器设置为`ip.addr == 127.0.0.1 and (tcp.srcport == 3000 or tcp.dstport == 3000) and http`，我们便可以捕获到端口3000且为http协议的数据包：



2. 数据分析

- **TCP协议三次握手建立连接**

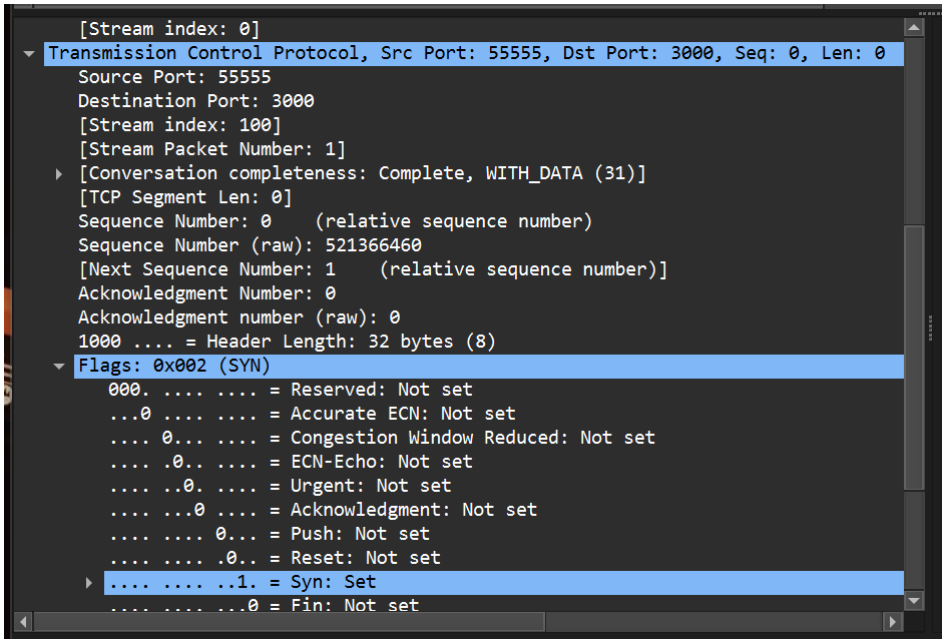
  TCP（三次握手）是一种可靠的传输控制协议，它使用**三次握手**（Three-Way Handshake）来建立连接，确保双方可以可靠地通信。这三次握手的过程涉及客户端和服务器之间的三个主要步骤。



我们将过滤条件改为捕获TCP协议的数据包，可以看到三次握手的全过程：

No.	Time	Source	Destination	Protocol	Length	Info
1031	71.004901	127.0.0.1	127.0.0.1	TCP	56	55555 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1032	71.004944	127.0.0.1	127.0.0.1	TCP	56	3000 → 55555 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1033	71.004987	127.0.0.1	127.0.0.1	TCP	44	55555 → 3000 [ACK] Seq=1 Ack=1 Win=2161152 Len=0

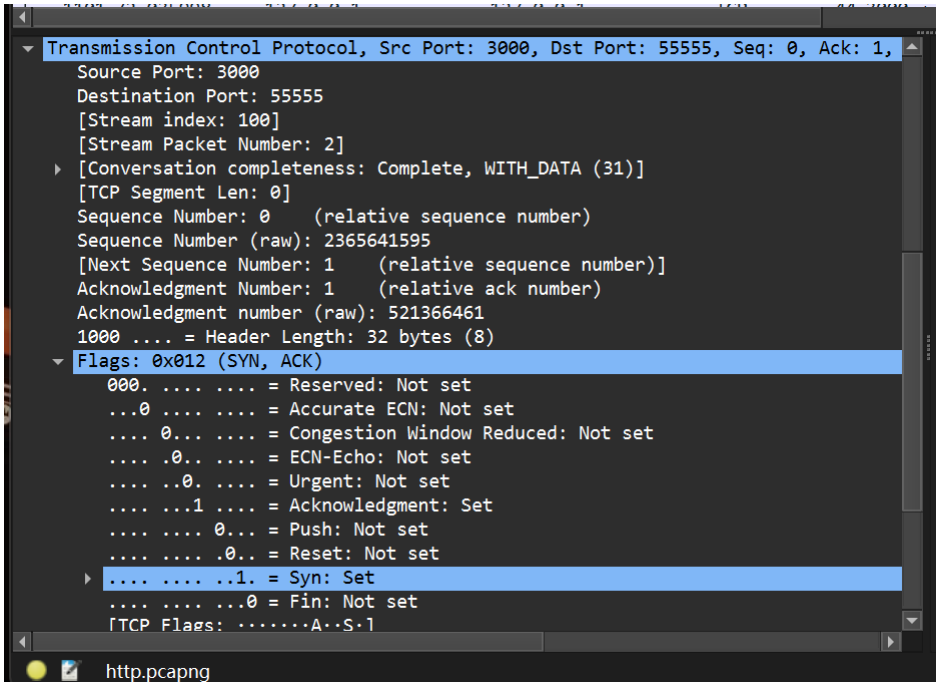
1. 第一次握手：客户端发送 SYN



- 客户端向服务器发送一个 **SYN**（Synchronize Sequence Number）报文段，用于请求建立连接。
- 客户端进入 **SYN-SENT** 状态，等待服务器的回应。
- **每个数据段的详细分析：**
  - **Source Port: 55555**：源端口为 `60611`，这是客户端随机选择的端口号，用于标识这个连接。
  - **Destination Port: 3000**：目标端口为 `3000`，表示客户端希望连接服务器上的端口 `3000`。这是服务器监听的端口，通常用于某个服务（如 Web 服务器）。
  - **Sequence Number: 0**：序列号为 `0`（相对序列号），实际值是 `3645251423`，这个序列号是客户端生成的一个随机初始序列号。序列号用于在数据传输过程中标识每个数据段的顺序。
  - **Acknowledgment Number: 0**：确认号为 `0`，因为这是连接的初始请求，客户端还没有接收到服务器的任何数据，所以确认号未设置。

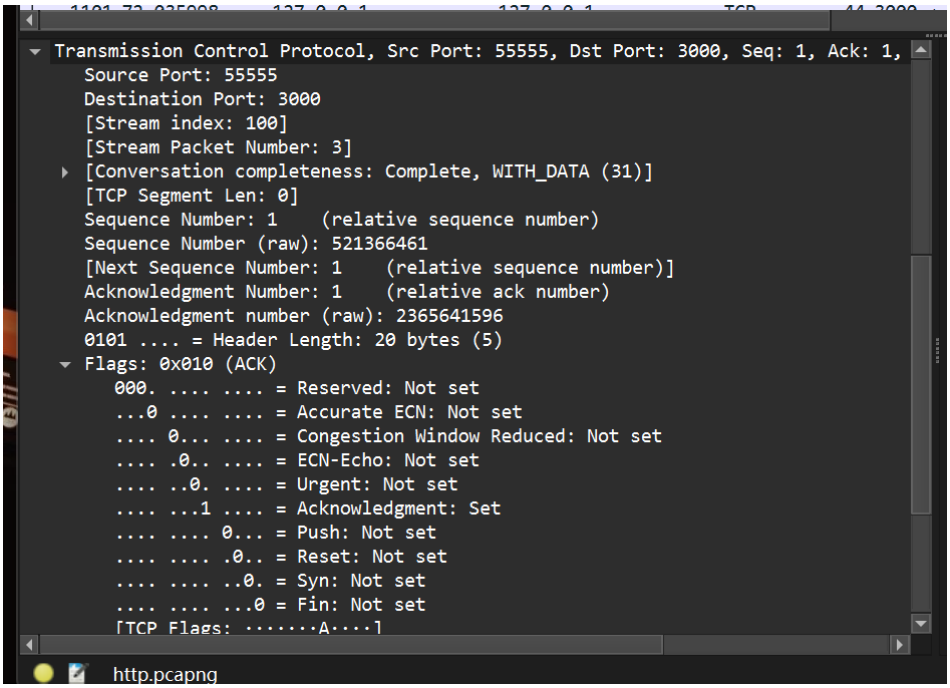
- **Header Length: 32 bytes (8)**：TCP 头部长度为 32 字节，通常包括基本 TCP 头部和一些可选字段或选项。
- **Flags: 0x002 (SYN)**：此标志位被设置为 `1`，表示这是一个同步请求，用于建立连接。SYN 位的设置是三次握手的第一步。其他标志位都为 `0`，说明这是一个单纯的连接请求，没有其他操作。

2. 第二次握手：服务器回复 SYN + ACK



- 服务器接收到客户端的 SYN 请求后，确认可以建立连接，于是向客户端发送一个 SYN + ACK 报文段。
- 服务器进入 **SYN-RECEIVED** 状态，等待客户端的确认。
- **每个数据段的详细分析**：
  - **Source Port: 3000**：源端口为 `3000`，表示服务器的端口号。服务器在监听端口 `3000`，并使用此端口来响应客户端的请求。
  - **Destination Port: 55555**：目标端口为 `60611`，这是客户端的源端口号。服务器回复数据包到这个端口，表示与客户端的通信仍保持在原连接上。
  - **Sequence Number: 0**：序列号为 `0`（相对序列号），实际的序列号值为 `4088581054`。这是服务器的初始序列号，用于确认接下来的通信顺序。此处的序列号是服务器随机生成的一个数值，后续会以此为基准进行数据传输。
  - **Acknowledgment Number: 1**：确认号为 `1`，实际确认号为 `3645251424`，表示服务器已经收到客户端的 `SYN` 报文，并期望接收的下一个序列号是 `1`。此确认号是客户端初始序列号 `3645251423 + 1`，用于告诉客户端服务器已经成功接收了第一个 `SYN` 报文。
  - **Header Length: 32 bytes (8)**：TCP 头部长度为 32 字节，说明可能包含了 TCP 选项部分。
  - **Flags: 0x012 (SYN, ACK)**：
    - **SYN**（Synchronize）：此标志位设置为 `1`，表示服务器响应客户端的连接请求。
    - **ACK**（Acknowledgment）：此标志位设置为 `1`，表示这是一个确认数据包，服务器在响应客户端的请求并确认已经收到。其他标志位（如 URG、PSH、FIN 等）都为 `0`，表示这仅是一个 `SYN + ACK` 确认连接的包，不包含其他操作。
  - **TCP Flags**：详细的标志位显示 `SYN` 和 `ACK` 位被设置为 `1`，表示这是三次握手中的第二步，用于确认连接并同步序列号。

3. 第三次握手：客户端回复 ACK

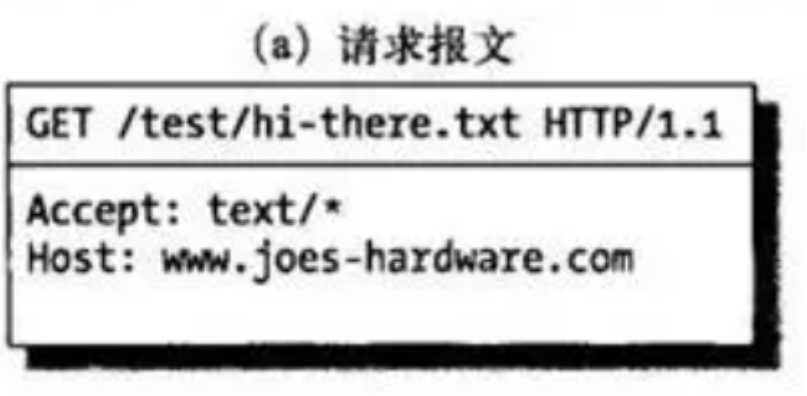


- 客户端收到服务器的 SYN + ACK报文后，向服务器发送一个 ACK 报文，确认连接建立。
- 客户端进入 **ESTABLISHED** 状态，表示连接已建立，可以开始传输数据。
- 服务器收到 ACK 报文后，也进入 **ESTABLISHED** 状态，连接建立完成。
- **每个数据段的详细分析：**
  - **Source Port: 60611：**源端口为 `60611`，这是客户端的端口，用于标识客户端的连接。
  - **Destination Port: 3000：**目标端口为 `3000`，这是服务器的端口号，服务器在该端口上接收客户端的请求。
  - **Sequence Number: 1：**序列号为 `1`（相对序列号），实际值为 `3645251424`。这个序列号是客户端在上一次握手中发送 `SYN` 时的初始序列号加 1（因为 `SYN` 消耗了一个序列号）。
  - **Acknowledgment Number: 1：**确认号为 `1`（相对确认号），实际值为 `4088581055`。这个确认号对应服务器的初始序列号加 1，表明客户端确认接收了服务器的 `SYN` 报文。这个确认号告诉服务器，客户端已经成功接收到服务器的 `SYN + ACK`。
  - **Header Length: 20 bytes：**TCP 头部长度为 20 字节，说明此报文没有包含 TCP 选项部分，只是一个简单的 ACK 确认。
  - **Flags: 0x010 (ACK)：**此标志位设置为 `1`，表示这是一个确认包。其他标志位（如 SYN、RST、FIN 等）均为 `0`，说明这是一个单纯的 ACK 确认包，不包含其他操作。

- **HTTP协议传输数据**

  HTTP（Hypertext Transfer Protocol）是一种用于客户端与服务器之间通信的协议。HTTP 协议有两种主要的报文：**请求报文**和**响应报文**。请求报文由客户端（如浏览器）发给服务器，用于请求资源；响应报文由服务器返回给客户端，用于提供请求的资源或状态信息。下面详细介绍 HTTP 请求报文和响应报文的结构和相关知识。

  1. **HTTP请求报文**



HTTP 请求报文由以下几部分组成：

- **请求行**

  请求行包括请求方法、请求资源的路径（URI）和 HTTP 版本号。格式如下：

  ```
  GET /index.html HTTP/1.1
  ```

  - **请求方法**：常见的请求方法包括：
    - `GET`：请求服务器发送指定资源。通常用于获取网页内容。
    - `POST`：向服务器提交数据，通常用于表单提交。
    - `PUT`：向服务器上传资源或更新资源。
    - `DELETE`：请求服务器删除指定资源。
    - `HEAD`：类似于 `GET`，但只请求头部信息，不返回资源主体。
    - 其他方法如 `OPTIONS`、`PATCH`、`TRACE` 等。
  - **请求路径**：资源在服务器上的路径（URI）。
  - **HTTP 版本**：如 `HTTP/1.1` 或 `HTTP/2`。

- **请求头部字段（Header）**

  请求头部字段包含了关于客户端和请求的元信息，每个头部字段占一行。常见的头部字段包括：

  - **Host**：指定服务器的主机名和端口号（如 `Host: example.com`）。
  - **User-Agent**：描述客户端的信息（如浏览器类型、版本等）。
  - **Accept**：客户端支持的响应内容类型（如 `text/html`、`application/json`）。
  - **Accept-Language**：客户端支持的语言（如 `zh-CN`、`en-US`）。
  - **Accept-Encoding**：客户端支持的编码方式（如 `gzip`、`deflate`）。
  - **Connection**：连接管理选项，如 `keep-alive` 表示保持连接。
  - **Content-Type**：请求主体的内容类型（如 `application/json`，通常用于 `POST` 请求）。
  - **Cookie**：包含客户端保存的 cookie 数据，服务器可以用来识别用户。

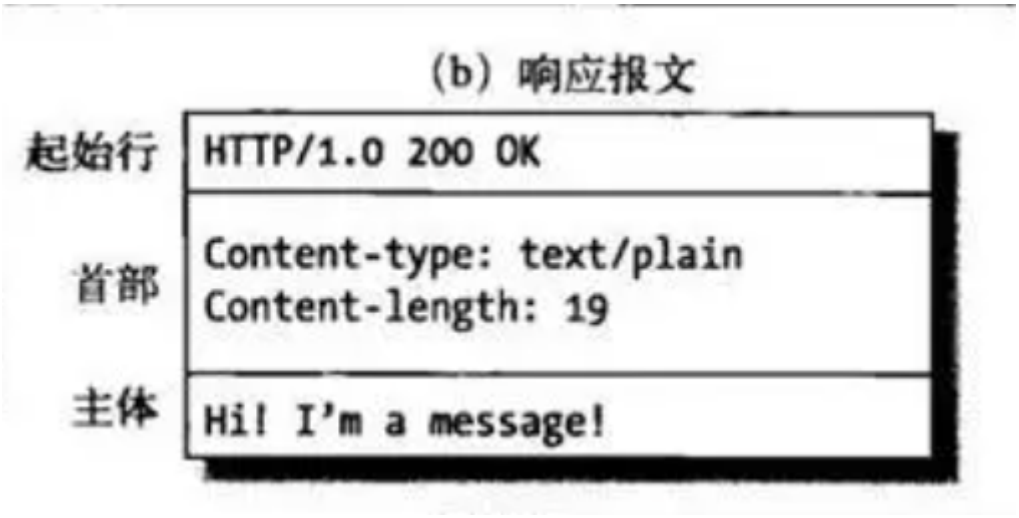
- **空行**

  空行用于分隔头部字段和请求主体。

- **请求主体（可选）**

  请求主体是可选的，通常在 `POST` 和 `PUT` 请求中使用，用于发送数据（如表单数据、文件上传内容）。请求主体的格式由 `Content-Type` 头部字段指定，如 `application/json`、`application/x-www-form-urlencoded` 等。

2. **HTTP响应报文**



HTTP 响应报文由以下几部分组成：

- **状态行**

  状态行包含 HTTP 版本、状态码和状态描述。格式如下：

  ```
  HTTP/1.1 200 OK
  ```

  - **HTTP 版本**：如 `HTTP/1.1` 或 `HTTP/2`。
  - **状态码**：三位数字表示请求的处理结果，分为以下几类：
    - `1xx`：信息性状态码，例如 `100 Continue`。
    - `2xx`：成功状态码，例如 `200 OK` 表示请求成功。
    - `3xx`：重定向状态码，例如 `301 Moved Permanently` 表示资源已永久移动。
    - `4xx`：客户端错误状态码，例如 `404 Not Found` 表示资源未找到。
    - `5xx`：服务器错误状态码，例如 `500 Internal Server Error` 表示服务器内部错误。
  - **状态描述**：对状态码的简单描述，例如 `OK`、`Not Found` 等。

- **响应头部字段（Header）**

  响应头部字段包含了关于服务器和响应的元信息。常见的响应头部字段包括：

  - **Date**：响应生成的日期和时间。
  - **Server**：服务器软件的信息（如 `Apache/2.4.1`）。
  - **Content-Type**：响应内容的 MIME 类型（如 `text/html`、`application/json`）。
  - **Content-Length**：响应主体的字节大小。
  - **Set-Cookie**：设置客户端存储的 cookie，用于用户识别或状态管理。
  - **Last-Modified**：资源的最后修改时间。
  - **ETag**：资源的唯一标识，用于缓存验证。
  - **Cache-Control**：缓存控制指令（如 `no-cache`、`max-age=3600`）。
  - **Expires**：资源的过期时间，用于缓存控制。

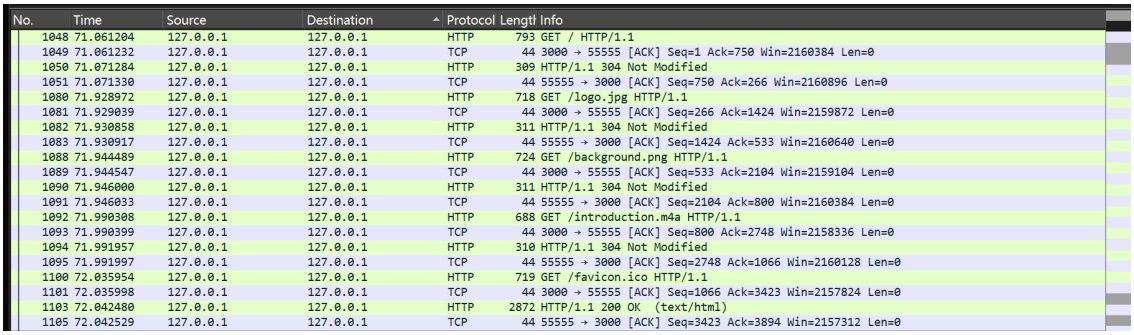
- **空行**

  空行用于分隔头部字段和响应主体。

- **响应主体（Body）**

  响应主体包含客户端请求的资源内容或返回的数据。对于 HTML 页面，响应主体就是 HTML 源代码。其他格式的响应主体可能是 JSON、XML、图片或文件等。

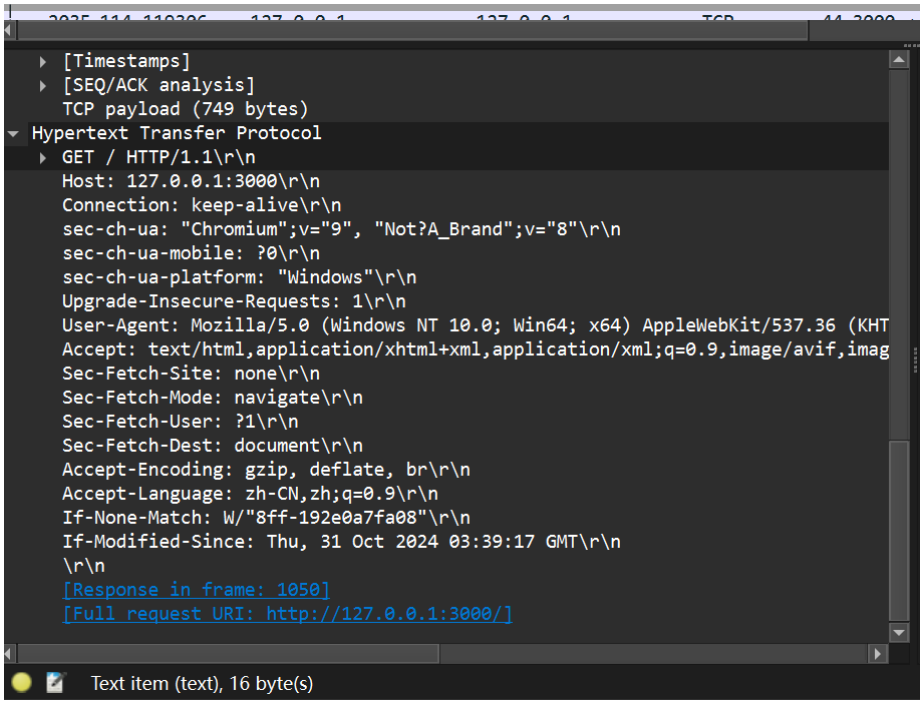
下面我们对http协议请求资源的具体过程进行分析。



No.	Time	Source	Destination	Protocol	Length	Info
1048	71.061204	127.0.0.1	127.0.0.1	HTTP	793	GET / HTTP/1.1
1049	71.061232	127.0.0.1	127.0.0.1	TCP	44	3000 → 55555 [ACK] Seq=1 Ack=750 Win=2160384 Len=0
1050	71.071284	127.0.0.1	127.0.0.1	HTTP	309	HTTP/1.1 304 Not Modified
1051	71.071330	127.0.0.1	127.0.0.1	TCP	44	55555 → 3000 [ACK] Seq=750 Ack=266 Win=2160896 Len=0
1080	71.928972	127.0.0.1	127.0.0.1	HTTP	718	GET /logo.jpg HTTP/1.1
1081	71.929039	127.0.0.1	127.0.0.1	TCP	44	3000 → 55555 [ACK] Seq=266 Ack=1424 Win=2159872 Len=0
1082	71.930858	127.0.0.1	127.0.0.1	HTTP	311	HTTP/1.1 304 Not Modified
1083	71.930917	127.0.0.1	127.0.0.1	TCP	44	55555 → 3000 [ACK] Seq=1424 Ack=533 Win=2160640 Len=0
1088	71.944489	127.0.0.1	127.0.0.1	HTTP	724	GET /background.png HTTP/1.1
1089	71.944547	127.0.0.1	127.0.0.1	TCP	44	3000 → 55555 [ACK] Seq=533 Ack=2104 Win=2159104 Len=0
1090	71.946000	127.0.0.1	127.0.0.1	HTTP	311	HTTP/1.1 304 Not Modified
1091	71.946033	127.0.0.1	127.0.0.1	TCP	44	55555 → 3000 [ACK] Seq=2104 Ack=800 Win=2160384 Len=0
1092	71.990308	127.0.0.1	127.0.0.1	HTTP	688	GET /introduction.m4a HTTP/1.1
1093	71.990399	127.0.0.1	127.0.0.1	TCP	44	3000 → 55555 [ACK] Seq=800 Ack=2748 Win=2158336 Len=0
1094	71.991957	127.0.0.1	127.0.0.1	HTTP	310	HTTP/1.1 304 Not Modified
1095	71.991997	127.0.0.1	127.0.0.1	TCP	44	55555 → 3000 [ACK] Seq=2748 Ack=1066 Win=2160128 Len=0
1100	72.035954	127.0.0.1	127.0.0.1	HTTP	719	GET /favicon.ico HTTP/1.1
1101	72.035998	127.0.0.1	127.0.0.1	TCP	44	3000 → 55555 [ACK] Seq=1066 Ack=3423 Win=2157824 Len=0
1103	72.042480	127.0.0.1	127.0.0.1	HTTP	2872	HTTP/1.1 200 OK (text/html)
1105	72.042529	127.0.0.1	127.0.0.1	TCP	44	55555 → 3000 [ACK] Seq=3423 Ack=3894 Win=2157312 Len=0

从图中可以看出，客户端总共向服务器发出了五次请求，服务器端返回了五次响应，分别向服务器请求了页面，logo等资源。

- **客户端HTTP协议请求**



请求行

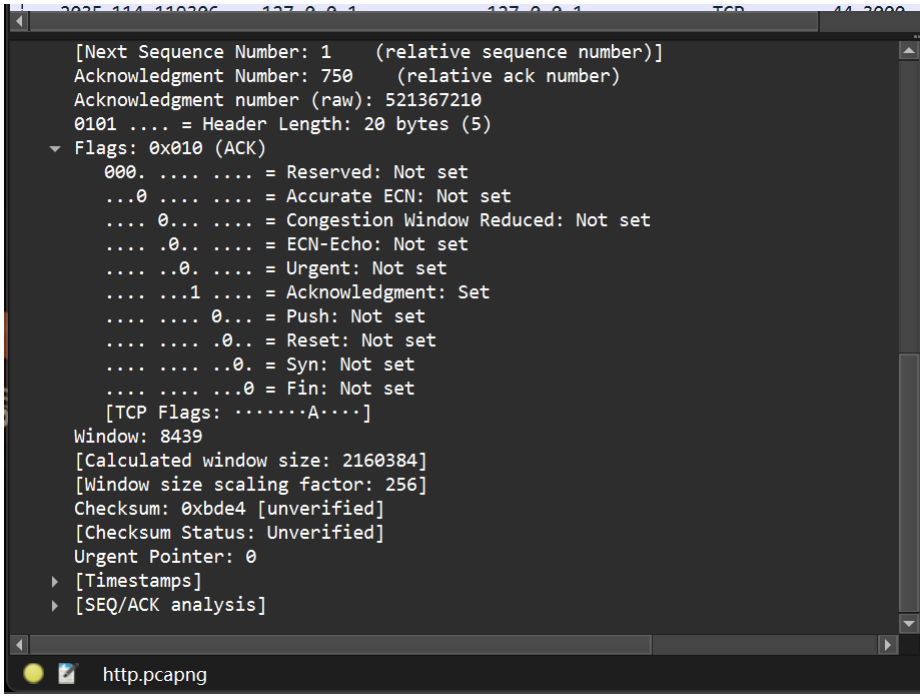
- **GET**：请求方法，表示客户端请求服务器的资源。
- **/**：请求的资源路径，这里是根路径（即服务器的主页或 `index.html`）。
- **HTTP/1.1**：协议版本，表示使用的是 HTTP/1.1 协议。

请求头部字段

以下是各个请求头部字段的作用：

- **Host: 127.0.0.1:3000**
  - 指定请求的服务器地址和端口号。在此例中，客户端请求的是本地服务器 `127.0.0.1`，端口为 `3000`。
- **Connection: keep-alive**
  - 表示客户端希望保持连接（`keep-alive`），即在响应之后不立即关闭连接，以便复用该连接来发送后续请求。
- **Cache-Control: max-age=0**
  - 指定客户端希望获取最新资源，将缓存的最大年龄设置为 `0` 秒，表示客户端请求不要使用缓存，而是直接从服务器获取最新的内容。
- **Upgrade-Insecure-Requests: 1**

        - 表示客户端支持将不安全的 HTTP 请求升级到 HTTPS 请求。这在请求 HTTPS 站点时更为常见，当前请求是 HTTP 协议，所以不会触发升级。
    - **User-Agent**
        - 指示客户端的信息，包括浏览器类型、操作系统版本等。例如，这里显示的是 `Mozilla/5.0`，浏览器为 `Chrome 91`，操作系统为 `Windows 10`。
    - **Accept**
        - 指示客户端可以接受的内容类型。这里指定客户端接受 HTML、XHTML、XML，以及支持图片格式 `image/avif` 和 `image/webp` 等。
    - **Accept-Encoding**
        - 指示客户端支持的内容编码方式。这里表示客户端可以接受 `gzip`、`deflate` 和 `br`（Brotli）压缩方式。
    - **Accept-Language**
        - 指示客户端的语言偏好，这里是 `zh-CN`（简体中文），其次是 `zh`（中文），客户端可能会按此顺序偏好接收语言版本。
- **服务器端回复ACK表示收到请求**

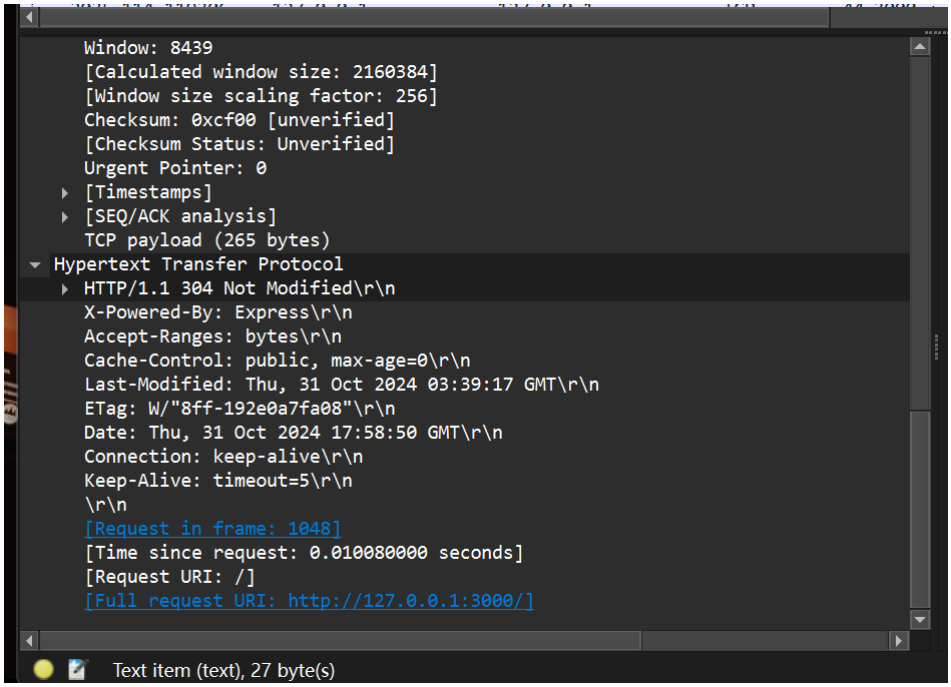


在 TCP 协议中，**ACK（Acknowledgment）位被设置为 Set** 表示该数据包包含一个**确认号（Acknowledgment Number）**，并确认已接收到之前发送的数据包。

- **服务器端HTTP协议响应**

服务器接收到客户端的 HTTP 请求后，进行以下操作：

1. **解析请求**：服务器解析请求的 URL 和头部信息，识别请求的资源路径。
2. **查找资源**：服务器检查请求的资源是否存在并具有访问权限。
3. **检查缓存条件**：如 `If-Modified-Since` 条件，检查资源是否自上次请求后被修改。
4. **生成响应：**
    - 如果资源没有被修改，返回 `304 Not Modified`，表示客户端可以使用缓存。
    - 如果资源已被修改或不在缓存中，返回 `200 OK`，并附带最新的资源内容。



状态行

- **HTTP/1.1**：表示使用的 HTTP 协议版本是 1.1。
- **304 Not Modified**：这是 HTTP 状态码，表示资源未修改。客户端可以使用本地缓存中的副本，而不必重新下载该资源。

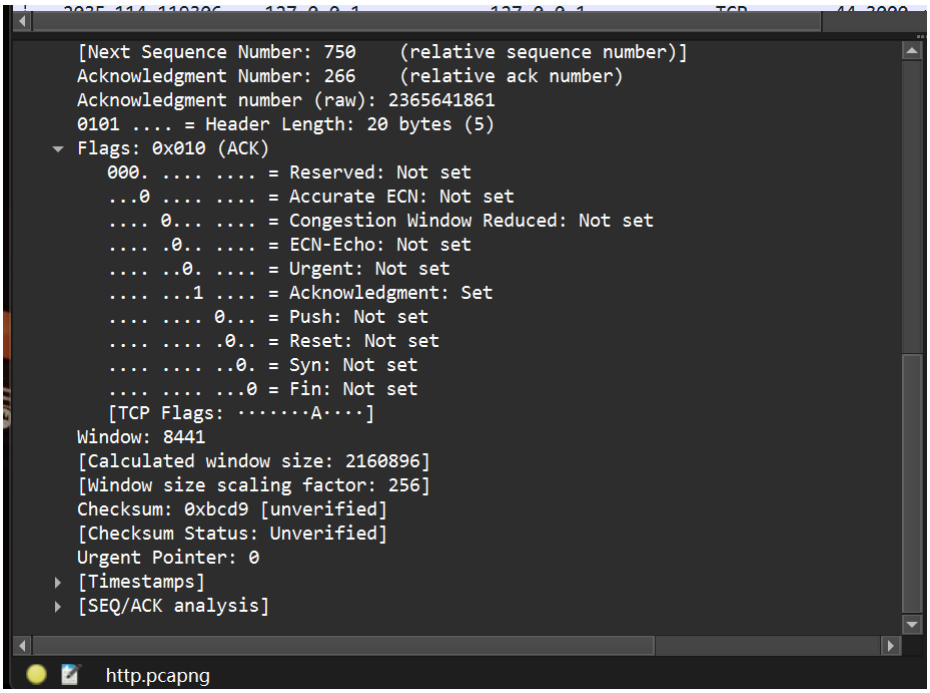
响应头部字段

以下是各个头部字段的作用：

- **X-Powered-By: Express**
  - 表示服务器使用了 Express 框架。`X-Powered-By` 是一个自定义头部，通常用来标识服务器的技术栈。
- **Accept-Ranges: bytes**
  - 表示服务器支持范围请求，客户端可以请求部分内容（如文件的特定字节范围）。这对于下载大文件或断点续传很有用。
- **Cache-Control: public, max-age=0**
  - **public**：指示响应可以被任何缓存保存（包括浏览器缓存和代理服务器缓存）。
  - **max-age=0**：指定资源的最大缓存时间为 0 秒，这意味着资源需要在每次请求时验证是否已修改。
- **Last-Modified: Thu, 31 Oct 2024 03:39:17 GMT**
  - 表示资源的最后修改时间。客户端可以将此值与本地缓存的资源进行比较，判断资源是否有更新。
- **ETag: W/"8ff-192e0a7fa08"**
  - `ETag` 是资源的唯一标识符，通过 ETag 可以判断资源是否有更新。客户端可以在请求时带上 ETag 值，以便服务器验证资源是否已更改。
- **Date: Thu, 31 Oct 2024 06:43:47 GMT**
  - 表示响应生成的时间，通常用于同步客户端和服务器的时间。
- **Connection: keep-alive**
  - 指定连接状态为 `keep-alive`，表示服务器希望在响应发送后保持 TCP 连接打开，以便处理后续请求。
- **Keep-Alive: timeout=5**

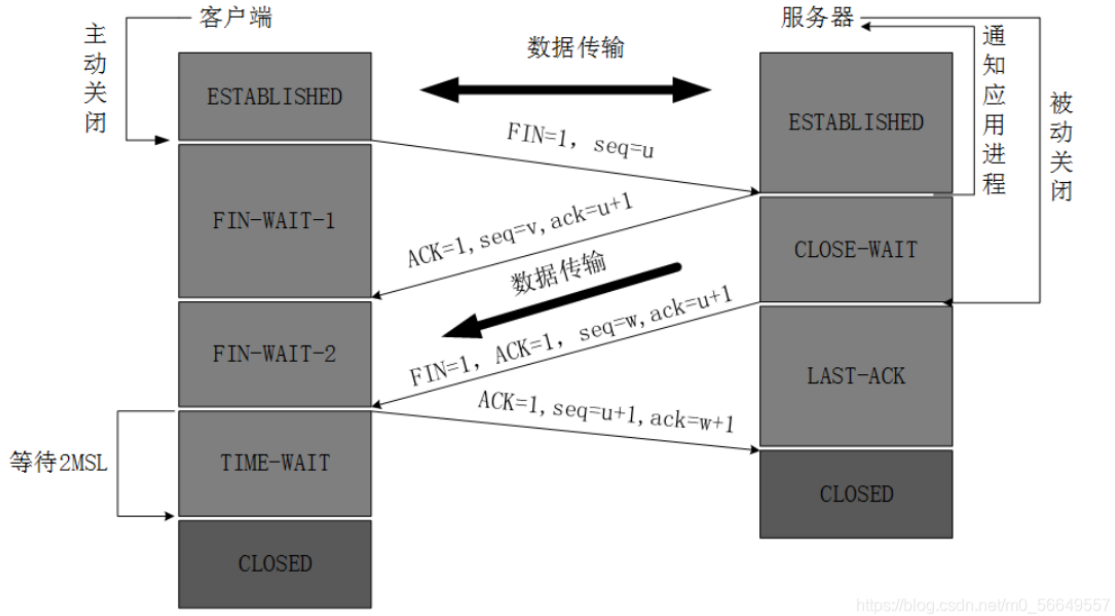
- 表示 `keep-alive` 连接的超时时间为 5 秒。如果在此时间内没有新的请求，服务器将关闭连接。

- **客户端回复ACK表示收到请求**

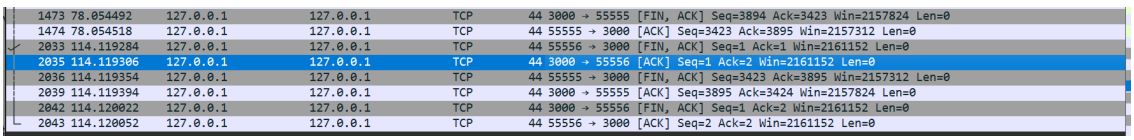


- **TCP协议四次挥手关闭连接**

  建立TCP连接需要三次握手，终止TCP连接需要四次挥手。

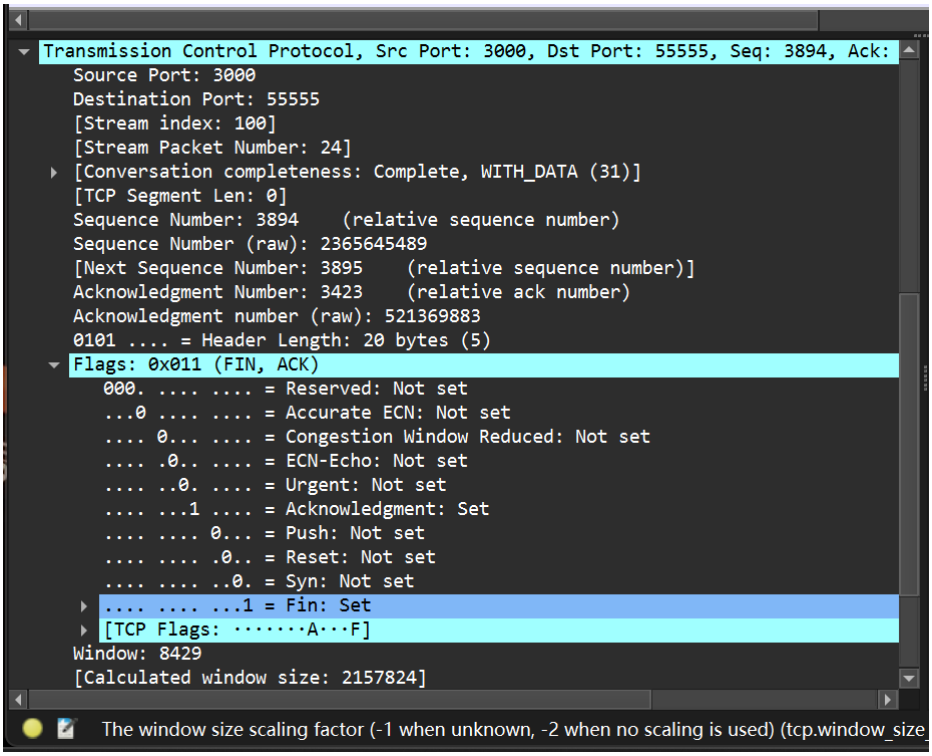


当我们关闭页面后我们可以看到正常完成四次挥手，总共完成了两个四次挥手是因为一次是关闭页面请求，一次关闭浏览器请求：



- **第一次挥手：服务器发送FIN包**

  在 TCP 四次挥手的第一次握手中，服务器端发送 `FIN` 包，并设置 `ACK` 位，表示已经接收完客户端发送的数据并请求关闭连接。通过 `FIN` 位的设置，服务器端表明不再发送数据但可以接收来自客户端的剩余数据。



源端口：3000

- 表示该数据包是从服务器端的 3000 端口发出的。

目标端口：55555

- 表示数据包的接收端口是客户端的 55555 端口。

Sequence Number（序列号）：3894

- 表示该数据包的序列号。
- 序列号是数据流中的一个编号，用于保证数据包的顺序传递。

Acknowledgment Number（确认号）：3423

- 表示服务器已经接收到从客户端发来的所有数据，并期待下一个数据包的序列号为 3423。
- 确认号在此阶段用于确认之前的数据传输完成。

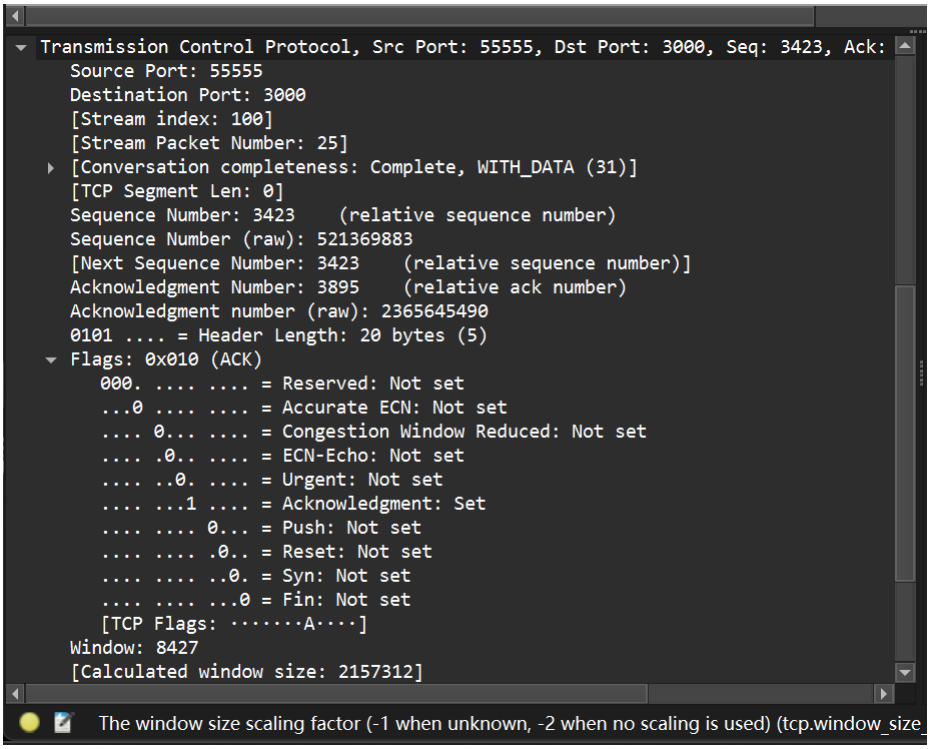
Flags：

- `FIN` 位设置为 `1`，表示请求关闭连接。
- `ACK` 位设置为 `1`，表示对先前数据的确认。

- **第二次挥手：客户端响应 ACK 包**

在 TCP 四次挥手过程中，第二次挥手是客户端收到服务器的关闭请求（FIN 包）后，返回一个 ACK 确认包来响应服务器端的关闭请求。此时：

- **客户端进入半关闭状态**：客户端已确认服务器的关闭请求，但连接的另一方向仍保持打开，客户端可能仍有数据需要发送。
- **ACK 位被设置**：确认号表示客户端已成功接收服务器的 FIN，并将等待服务器进一步的关闭动作。



源端口：55555

- 这是客户端的端口号，表明数据包是从客户端发出的。

目标端口：3000

- 这是服务器的端口号，表明数据包是发给服务器的。

标志位（Flags）：`0x010 (ACK)`

- 只有 **ACK** 位被设置为 1，表示客户端确认收到服务器的 FIN 包。
- 没有设置 FIN 位，因为客户端仅在此包中发送确认，而并未准备关闭连接。

Sequence Number（序列号）：3423

- 这是客户端发送的序列号，用于标识当前数据包的顺序。
- 在 ACK 包中，序列号通常不会改变，它只是一个用于标识的数值。

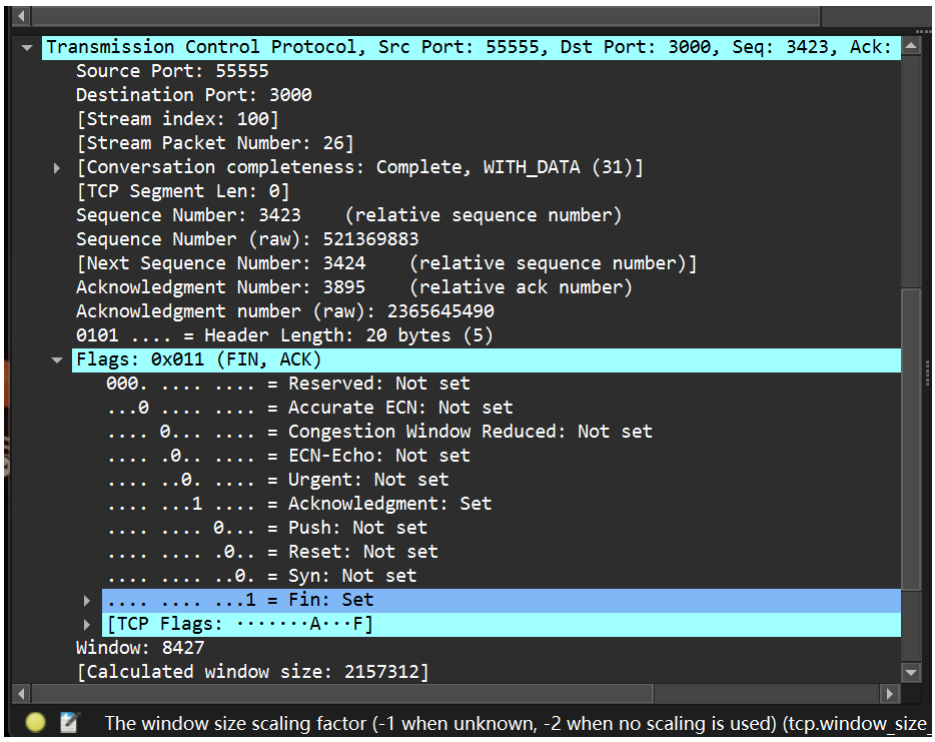
Acknowledgment Number（确认号）：3895

- 这是客户端的确认号，表示客户端已收到客户端的 FIN 包，并确认了从服务器发来的所有数据。
- 确认号 `3895` 表示客户端已经收到服务器的最后一个序列号为 `3894` 的数据包，期待服务器的下一个序列号为 `3895`（即没有更多数据传输）。

- **第三次挥手：客户端发送 FIN 包**

在 TCP 四次挥手的第三次步骤中，客户端向服务器发送一个带有 `FIN, ACK` 标志的数据包，告知服务器：

- **客户端准备关闭连接**，不再发送数据。
- **确认接收到客户端的 FIN 请求**，这是关闭连接的第二次确认。

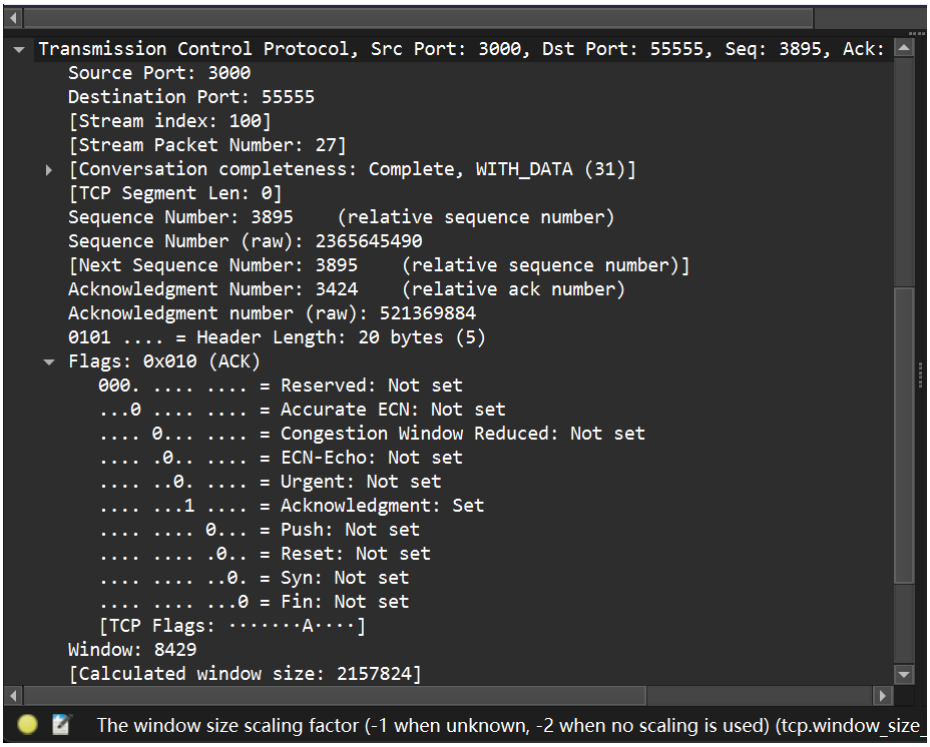


- **源端口**：55555
  - 客户端的端口，表明这个数据包从客户端发出。
- **目标端口**：3000
  - 服务器的端口，表明数据包发往服务器。
- **标志位（Flags）**：`0x011 (FIN, ACK)`
  - 这里设置了两个标志位：
    - **FIN**：表示客户端请求关闭连接。
    - **ACK**：确认服务器之前发来的 FIN 包。
  - `FIN` 位的设置表明客户端在完成自己的数据传输后，通知服务器它也要关闭连接。
  - `ACK` 位的设置表明客户端已确认客服务器发送的 FIN 包（第二次挥手中的 FIN 请求）。
- **Sequence Number（序列号）**：3423
  - 这是客户端发送的序列号，用于标识当前数据包的顺序。
  - 序列号表明这是客户端在关闭连接前发送的最后一个数据包。
- **Acknowledgment Number（确认号）**：3895
  - 客户端的确认号，表示客户端已经确认接收到了服务器的 FIN 包。
  - 确认号 `3895` 表明客户端接收了服务器发来的数据和 FIN 请求，并准备关闭连接。

- **第四次挥手：服务器发送 ACK 包，确认客户端的 FIN 包**

  在 TCP 四次挥手的第四次步骤中，服务器向客户端发送一个带有 ACK 的数据包，告知客户端：

  - **服务器已确认客户端的关闭请求**，这是四次挥手过程的最后一步。
  - 此时，客户端可以关闭连接。



- **源端口**：3000
  - 服务器的端口，表明这个数据包是从服务器发出的。
- **目标端口**：55555
  - 客户端的端口，表明数据包发往客户端。
- **标志位（Flags）**：`0x010 (ACK)`
  - 只有 **ACK** 标志位被设置为 1，表示服务器确认收到了客户端的 FIN 包。
  - 此时没有设置 FIN 位，因为客户端已经完成了对服务器 FIN 请求的确认，连接即将完全关闭。
- **Sequence Number（序列号）**：3895
  - 服务器发送的序列号，用于标识当前数据包的顺序。
  - 序列号表明这是服务器关闭连接前的最后一个数据包。
- **Acknowledgment Number（确认号）**：3424
  - 服务器的确认号，表示已经收到客户端的 FIN 包（第三次挥手中的 FIN 请求），并确认从客户端发来的所有数据。
  - 确认号 `3424` 表示服务器收到了客户端的最后一个数据包，并表示连接的关闭请求已被确认。
- **窗口大小（Window）**：8429
  - 服务器的接收窗口大小，此时窗口大小已经没有实际意义，因为连接即将关闭。

心得感悟

在这次实验中，通过wireshark分析和捕获 TCP 协议的连接建立与关闭过程，学会了wiresharkshark中抓包的基本操作，并且深入理解了 TCP **三次握手**，**http请求与响应**和**四次挥手**的原理。每一个握手和挥手的数据包都包含了特定的标志位、序列号和确认号，确保了数据的可靠传输和连接的正常关闭。通过使用 Wireshark 对这些数据包进行分析，我看到了客户端和服务器之间的交互细节，以及如何通过四次挥手安全地关闭连接。实验中，尤其是在分析每一步的数据包时，让我体会到网络协议的**严谨性**和**复杂性**。TCP 的设计不仅仅是实现数据传输，它更强调**可靠性**，保证数据传输过程中的**顺序和完整性**。四次挥手机制也展示了如何在不丢失数据的前提下，优雅地关闭连接，避免资源浪费。通过这次实验，我对网络通信的底层原理有了更清晰的理解，这对我今后编写网络应用和优化网络性能有很大帮助。