实验一：分析HTTP协议

（原为个人信息，此处隐藏）

一、实验目的

- 利用 Wireshark 软件分析 **HTTP** 及其下层协议（**TCP 协议**）；
- 了解网络中**数据封装**的概念；
- 掌握 **HTTP** 及 **TCP** 协议的工作过程。

二、实验内容

- 启动 Wireshark 软件，进行报文截获；
- 在浏览器访问 http://www.xjtu.edu.cn 页面（打开网页，浏览并关闭页面）；
- 停止 Wireshark 的报文截获，分析截获报文。

三、实验要求

- 从截获的报文中选择 **HTTP** *请求报文*（即 **GET** 报文）和 **HTTP** *应答报文*，并分析各字段的值；
- 综合分析截获的报文，概括 **HTTP** 协议的工作过程；
- 从截获报文中选择 **TCP** 建立连接和释放连接的报文，分析各个字段的值并概括 **TCP 协议**的工作过程。

四、实验相关理论简述

1. 数据封装

数据封装是网络通信中的一个核心概念，指的是在数据通过网络传输时，按照**分层协议模型（如 OSI 或 TCP/IP 模型）**的要求，在每一层对数据增加特定的控制信息（称为协议头或协议尾）的过程。封装后的数据逐层传递，直到通过物理介质传输到目标设备。数据封装发生在 **发送方设备**，大致过程如下：

- 用户生成的原始信息（如网页内容、电子邮件、文件等）被 **应用层** 协议处理（如 HTTP、FTP 等），产生的结果称为 **数据（Data）**；
- **传输层** 将应用层数据划分为更小的段，并附加控制信息（如端口号、校验和），产生的结果称为 **段（Segment）**；

- **网络层** 将传输层段封装成数据包，并附加 IP 地址等信息，产生的结果称为 **包（Packet）**；
- 数据链路层将网络层数据包封装成帧，并附加 MAC 地址、帧校验序列（FCS）等，产生的结果称为 **帧（Frame）**；
- 物理层：数据链路层的帧被转换为比特流（电信号、光信号或无线信号）通过物理介质传输，产生的结果称为**比特流（Bits）**。

数据到达接收方时，进行 **数据解封装**，这一过程类似于数据封装的逆操作。物理层将接收到的信号转换为比特流，数据链路层解析帧并提取网络层数据包，网络层提取传输层段，传输层提取应用层数据，最终交给应用程序处理。

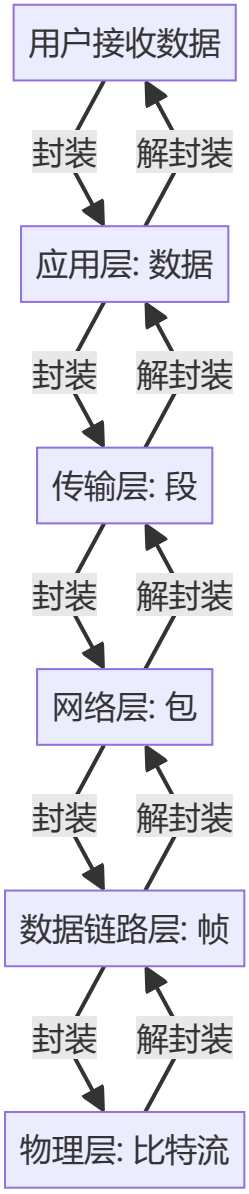


图1：数据封装与解封装流程

2. HTTP与TCP协议工作过程概览

HTTP（超文本传输协议）和**TCP**（传输控制协议）是现代网络通信中的核心协议。它们在 **应用层** 和 **传输层** 分别扮演着重要角色，确保了数据的可靠传输和有效的通信。

HTTP 是万维网的应用层协议，也是万维网的核心。它运行在 **TCP/IP** 之上，通过两个程序实现：一个是 **客户端（Client）** 程序（通常是浏览器），另一个是 **服务器（Server）** 程序（通常称为Web服务器）。两者通过交换**HTTP**报文来完成网页请求和响应。由于**HTTP**所基于的传输层协议**TCP**是 **面向连接** 的，可以提供**"可靠的"**文件传输（**不出错、不重复、不失序、不丢失**），最初的**HTTP/1.0**最初不提供对于连接的维持，每次在客户机接受完响应报文后TCP连接即告结束；也不存储任何有关客户机的状态信息，因此是**无连接、无状态**的协议；而**HTTP/1.1**对连接性作出了一定改进，在完成一次HTTP报文后还可以继续保持连接，即支持**持续连接**；另外，**HTTP/2**在这一持久连接上可以实现**请求管道化**，即客户机可以在一个持久的TCP连接中**并发**地发送多个请求。通过抓包分析得出，本实验访问目标网站使用的为**HTTP/1.1**协议。具体报文细节稍后会给出分析。

TCP是互联网和局域网中最常用的传输层协议之一。它属于 **面向连接** 的协议，保证数据传输的**可靠性**，并在两个主机之间建立 **端到端** 的连接。TCP协议的核心目标是确保数据能够 **不出错、不重复、不失序、不丢失** 从源主机传输到目标主机。为了实现这一目标，TCP使用了一系列机制来保证数据的可靠性、完整性和顺序性，我们形象地概括其为**"三次握手，四次挥手"**。具体细节我们稍后结合抓包结果分析。**TCP** 作为传输层协议，为 HTTP 提供可靠的数据传输通道。HTTP 本身并不处理数据传输的可靠性、顺序和完整性，它依赖于 TCP 来确保数据的可靠传送。每个 HTTP 请求和响应都通过一个 TCP 连接传输。无论是文本内容、图片，还是其他资源，都通过建立的 TCP 连接进行传输。

总结来说，**HTTP** 用于定义数据的格式和交换规则，而 **TCP** 则负责数据的可靠传输。两者密切配合，共同确保了 Web 应用的正常运行和数据的可靠传输。

在完成各种网络通信服务的时候，分别处于应用层和传输层的两个协议需要配合工作。为了允许用户访问传输服务，传输层必须为应用层提供 **传输服务接口**，而 **套接字Socket** 便是这样的一种接口。但是需要注意，**Socket 是应用程序与操作系统之间的接口**。当在程序中使用 Socket 原语时，操作系统会将这些原语转化为相应的底层网络操作（如创建连接、发送数据等）。这些操作通常在操作系统内部完成，具体的 API 调用本身不会在网络数据包中显示出来。抓包工具只能看到应用层数据如何通过网络传输，但不能看到应用程序内部如何调用这些原语。

从打开浏览器浏览目标网页，到关闭浏览器，工作过程可以粗略地分为以下几个步骤（包含了**DNS**查询过程）：

S1: 用户输入网址（域名）：如本实验的`http://www.xjtu.edu.cn`。

S2: 浏览器检查缓存：浏览器首先会检查本地缓存是否已有该域名的 IP 地址。如果有缓存，就直接使用缓存的 IP 地址，无需再进行 DNS 查询。

S3: DNS 查询：如果本地没有缓存，浏览器会向 DNS 服务器发送一个查询请求，查找该域名对应的 IP 地址。

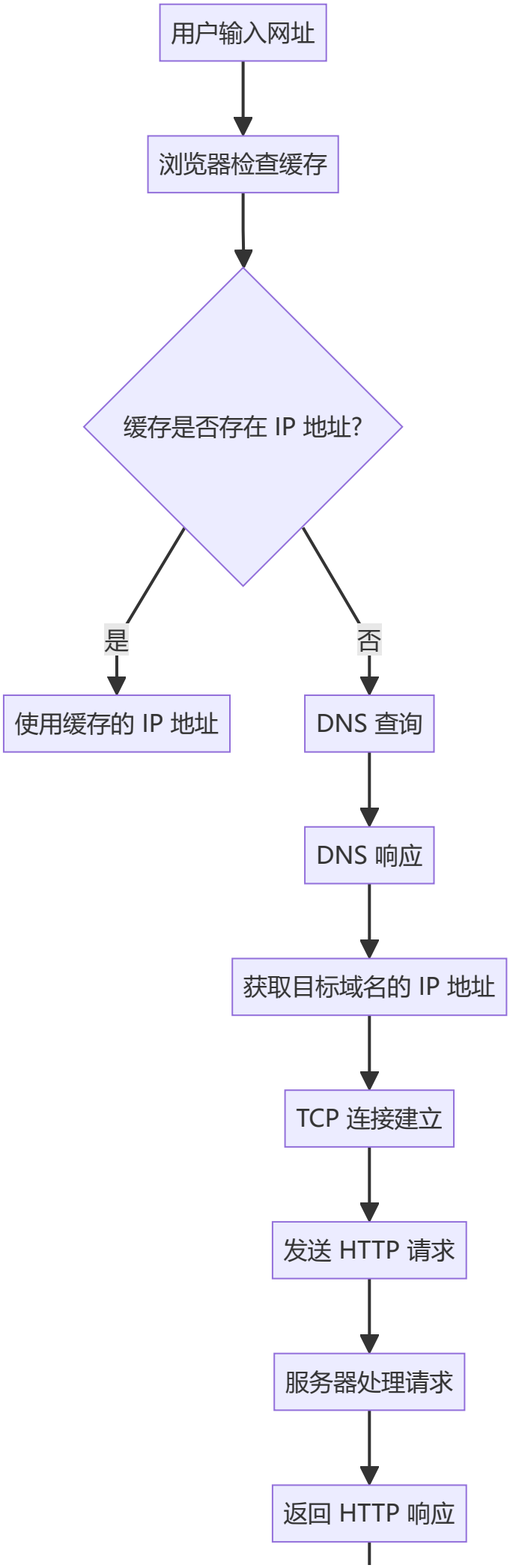
S4: DNS 响应：DNS 服务器返回目标域名对应的 IP 地址（如`202.117.1.13`）。

S5: TCP 连接建立：浏览器通过 TCP/IP 协议与目标服务器建立连接。此时，浏览器和服务器之间会通过三次握手建立可靠的连接。

S6: 发送 HTTP 请求：浏览器通过建立的 TCP 连接向服务器发送 HTTP 请求（例如 GET 请求）。

S7: 服务器响应：服务器处理请求并返回 HTTP 响应。

S7: 关闭连接：请求和响应完成后，若 HTTP 版本为 1.0 或无持久连接，连接会立即关闭。


用户输入网址
浏览器检查缓存
缓存是否存在 IP 地址?
是
否
使用缓存的 IP 地址
DNS 查询
DNS 响应
获取目标域名的 IP 地址
TCP 连接建立
发送 HTTP 请求
服务器处理请求
返回 HTTP 响应

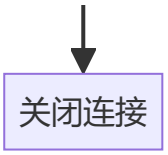


图2：基本工作流程

这一过程中有关**HTTP**与**TCP**协议工作的细节流程，在实验结果分析时会结合抓包结果更加详细地说明。

五、实验结果分析

1. DNS查询

本实验电脑通过连接**Wi-Fi**网络，以**无线局域网（WLAN）**接入网络。所以**路由器**会充当**网关**以及**本地DNS服务器**，首先查询本地缓存中有无待解析域名对应的IP地址，决定返回缓存中的信息或者将查询请求转发到公共DNS服务器递归查询。我们可以在命令行窗口中查询相关信息，输入`ipconfig /all`，所得结果如下：

```
Windows IP 配置

   主机名  . . . . . . . . . . . . . . : DESKTOP-TLBKH7R
   主 DNS 后缀 . . . . . . . . . . . . :
   节点类型  . . . . . . . . . . . . . : 混合
   IP 路由已启用 . . . . . . . . . . . : 否
   WINS 代理已启用 . . . . . . . . . . : 否

以太网适配器 以太网:

   媒体状态  . . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . . :
   描述. . . . . . . . . . . . . . . . : Realtek PCIe GbE Family
Controller
   物理地址. . . . . . . . . . . . . . : 04-BF-1B-6F-95-40
   DHCP 已启用 . . . . . . . . . . . . : 是
   自动配置已启用. . . . . . . . . . . : 是

无线局域网适配器 本地连接* 1:

   媒体状态  . . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . . :
   描述. . . . . . . . . . . . . . . . : Microsoft Wi-Fi Direct Virtual
Adapter
   物理地址. . . . . . . . . . . . . . : DC-46-28-11-EB-59
   DHCP 已启用 . . . . . . . . . . . . : 是
   自动配置已启用. . . . . . . . . . . : 是
```

```
无线局域网适配器 本地连接* 2:

   媒体状态  . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . :
   描述. . . . . . . . . . . . . . . : Microsoft Wi-Fi Direct Virtual
Adapter #2
   物理地址. . . . . . . . . . . . . : DE-46-28-11-EB-58
   DHCP 已启用 . . . . . . . . . . . : 否
   自动配置已启用. . . . . . . . . . : 是

无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . . . :
   描述. . . . . . . . . . . . . . . : Intel(R) Wi-Fi 6 AX201 160MHz
   物理地址. . . . . . . . . . . . . : DC-46-28-11-EB-58
   DHCP 已启用 . . . . . . . . . . . : 是
   自动配置已启用. . . . . . . . . . : 是
   本地链接 IPv6 地址. . . . . . . . : fe80::d3a:1f6e:571c:14e0%8(首选)
   IPv4 地址 . . . . . . . . . . . . : 192.168.2.44(首选)  #此处注明本机的
IP地址
   子网掩码  . . . . . . . . . . . . : 255.255.255.0
   获得租约的时间  . . . . . . . . . : 2024年12月17日 14:12:48
   租约过期的时间  . . . . . . . . . : 2024年12月18日 14:12:48
   默认网关. . . . . . . . . . . . . : 192.168.2.1  #此处注明网关的地址
   DHCP 服务器 . . . . . . . . . . . : 192.168.2.1
   DHCPv6 IAID . . . . . . . . . . . : 165430824
   DHCPv6 客户端 DUID  . . . . . . . : 00-01-00-01-2C-28-D2-81-04-BF-1B-
6F-95-40
   DNS 服务器  . . . . . . . . . . . : 192.168.2.1  #说明网关同时也充当本地
DNS服务器
   TCPIP 上的 NetBIOS  . . . . . . . : 已启用

以太网适配器 以太网 2:

   媒体状态  . . . . . . . . . . . . : 媒体已断开连接
   连接特定的 DNS 后缀 . . . . . . . :
   描述. . . . . . . . . . . . . . . : Netease UU TAP-Win32 Adapter
V9.21
   物理地址. . . . . . . . . . . . . : 00-FF-F3-FC-85-85
   DHCP 已启用 . . . . . . . . . . . : 是
   自动配置已启用. . . . . . . . . . : 是
```

所以可以看到，本机 IPv4 地址为 `192.168.2.44`，网关地址为 `192.168.2.1`，同时也充当DNS服务器。所以我们假设开始的 DNS 查询就在本地 DNS 服务器进行，源IP地址和目标IP地址就应该是本机地址和网关地址。而校园网对于学校官网应该是有缓存的，所以可能不需要网关转发给公共 DNS 。下面提取真实抓包信息来加以验证以上猜想：

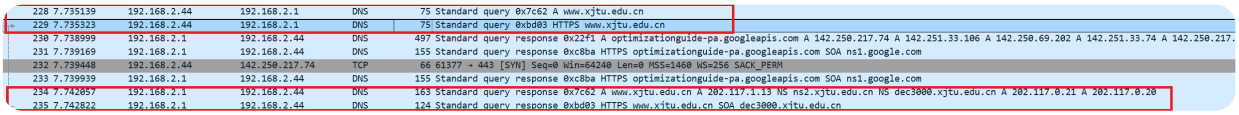


图3 有关DNS查询的部分报文

如图，上面两个编号为 `228` 与 `229` 的包就是用户机向网关发送的两个 `DNS标准查询（Standard query）`。一个带有 `A` 标识，一个带有 `HTTPS` 标识。前者用于解析域名到 `IPv4` 地址，是传统的DNS解析方式；后者是 `HTTPS` 记录，用于用于检查该域名是否支持 HTTPS 加速特性（*事实上在之前输入URL时不加 `http://` 协议头的时候，浏览器默认使用 `https://` 填充，于是在抓取报文时出现了许多协议名为 `TLSv1.3` 的网络包，而没有出现实验要求分析的 HTTP 报文*）。对应地，以下 `234` 和 `235` 就是对应两个查询请求的返回包。

`228` 包的解析结果如下：

```
Domain Name System (query)
    Transaction ID: 0x7c62
  Flags: 0x0100 Standard query
      0... .... .... .... = Response: Message is a query
      .000 0... .... .... = Opcode: Standard query (0)
      .... ..0. .... .... = Truncated: Message is not truncated
      .... ...1 .... .... = Recursion desired: Do query recursively
      .... .... .0.. .... = Z: reserved (0)
      .... .... ...0 .... = Non-authenticated data: Unacceptable
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  Queries
    > www.xjtu.edu.cn: type A, class IN
    [Response In: 234]
```

图4 包228解析结果

Transaction ID：`0x7c62`

用于标识该请求的唯一标识符。客户端通过它与响应报文对应。

Flags：`0x0100`

这是一个 **标准查询** （Standard Query）。解析各标志位：

0... .... = Response：表示这是一个**请求**（值为 `0`，不是**响应**）。

.000 0... .... = Opcode：标准查询（值为 `0`）。

.... .0.. .... = Truncated：未被截断，数据完整。

.... ..1. .... = Recursion desired：表示客户端请求**递归查询**。

....0 = Non-authenticated data：未启动安全认证，当前查询报文是无身份验证的数据。

Questions: 1

当前查询中有 1 个问题，即只查询了一个域名。

Answer RRs: 0

资源记录（RRs）情况。当前报文中**没有返回**答案记录，因为这只是一个**查询报文**。

Authority RRs: 0

没有权威名称服务器信息。

Additional RRs: 0

没有额外的附加信息。

最后 `Queries` 做出总结。这个报文是一个典型的 **DNS 查询请求**，基本信息如下：

(1). 客户端向 DNS 服务器发送了一个请求，询问 `www.xjtu.edu.cn` 的 **A 记录**（IPv4 地址），资源记录属于 `IN (Internet)` 类别。
(2). 请求中带有 **递归查询标志位**（RD=1），表明客户端希望 DNS 服务器帮助完成递归解析。
(3). 当前报文是纯查询内容，**没有任何响应数据**，因为这是请求而非响应。

而 `229` 包的解析结果和前包极其类似，只不过在 `type` 处为 `HTTPS`，故省略不表。下面分析一下对应 `228` 号报文的 DNS 响应报文 `234`：

```
Transaction ID: 0x7c62
Flags: 0x8180 Standard query response, No error
    1... .... .... .... = Response: Message is a response
    .000 0... .... .... = Opcode: Standard query (0)
    .... .0.. .... .... = Authoritative: Server is not an authority for domain
    .... ..0. .... .... = Truncated: Message is not truncated
    .... ...1 .... .... = Recursion desired: Do query recursively
    .... .... 1... .... = Recursion available: Server can do recursive queries
    .... .... .0.. .... = Z: reserved (0)
    .... .... ..0. .... = Answer authenticated: Answer/authority portion was not authenticated by the server
    .... .... ...0 .... = Non-authenticated data: Unacceptable
    .... .... .... 0000 = Reply code: No error (0)
Questions: 1
Answer RRs: 1
Authority RRs: 2
Additional RRs: 2
Queries
    www.xjtu.edu.cn: type A, class IN
Answers
    www.xjtu.edu.cn: type A, class IN, addr 202.117.1.13
Authoritative nameservers
    xjtu.edu.cn: type NS, class IN, ns ns2.xjtu.edu.cn
    xjtu.edu.cn: type NS, class IN, ns dec3000.xjtu.edu.cn
Additional records
    ns2.xjtu.edu.cn: type A, class IN, addr 202.117.0.21
    dec3000.xjtu.edu.cn: type A, class IN, addr 202.117.0.20
[Request In: 228]
[Time: 0.006918000 seconds]
```

图5 包234解析结果

部分基本信息：

Flags： `0x8180`

解析标志位：

1... .... = Response：这是一个响应报文。

.000 0... .... = Opcode：标准查询（0）。

.... .1.. .... = Recursion desired：客户端请求递归查询。

.... ..1. .... = Recursion available：服务器支持递归查询。

.... = Reply code：**No Error**，表示查询成功。

回答部分（Answers）返回了 A 记录（IPv4 地址）：

Name：`www.xjtu.edu.cn`

Type：`A`（IPv4 地址）。

Class：`IN`（Internet 类别）。

TTL（生存时间）：`1256` 秒（约 21 分钟）。

Address：`202.117.1.13`，这是 `www.xjtu.edu.cn` 的 IPv4 地址。

权威名称服务器部分（Authority RRs）提供了域名 `xjtu.edu.cn` 的权威名称服务器：

a. **Name**：`xjtu.edu.cn`

Type：`NS`（Name Server）。

Name Server：`ns2.xjtu.edu.cn`

TTL：`838` 秒（约 13 分钟）。

b. **Name**：`xjtu.edu.cn`

Type：`NS`（Name Server）。

Name Server：`dec3000.xjtu.edu.cn`

TTL：`838` 秒（约 13 分钟）。

说明：这些是负责管理 `xjtu.edu.cn` 域名的权威名称服务器。

附加记录部分（Additional RRs）：

提供了权威名称服务器的 IPv4 地址，分别为 `ns2.xjtu.edu.cn (Type A, Address 202.117.0.21)`，`dec3000.xjtu.edu.cn (Type A, Address 202.117.0.20)`。这些附加记录帮助客户端直接联系权威名称服务器，减少后续查询延迟。

从上图可以得出以下结论：

(1). 客户端查询了 `www.xjtu.edu.cn` 的 **A 记录**。

(2). 服务器响应成功，返回了 `www.xjtu.edu.cn` 的 IPv4 地址：`202.117.1.13`。

(3). 同时提供了 `xjtu.edu.cn` 的权威名称服务器信息（`ns2.xjtu.edu.cn` 和 `dec3000.xjtu.edu.cn`）。

(4). 附加记录中提供了权威服务器的具体 IP 地址，便于后续的直接查询。

2. TCP与HTTP协议工作原理分析

数据预处理

因为研究重点在于**HTTP**以及**TCP**协议的相关细节，而在开始抓包后信息过于繁杂，不利于提取相关信息，所以要对抓包信息做一些筛选。通过分析，我们可以得出了目标网站的 IPv4 地址是`202.117.1.13`，所以设置筛选条件为只展示源/目标 IP 地址为此的报文。在筛选条件窗口输入命令`ip.src_host==202.117.1.13 or ip.dst_host==202.117.1.13`，获得结果如下（部分）：

```
   236 7.743099     192.168.2.44          202.117.1.13          TCP         66 61378 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
   237 7.743334     192.168.2.44          202.117.1.13          TCP         66 61379 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
   238 7.744919     202.117.1.13          192.168.2.44          TCP         66 80 → 61378 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1440 SACK_PERM WS=128
   239 7.744919     202.117.1.13          192.168.2.44          TCP         66 80 → 61379 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1440 SACK_PERM WS=128
   240 7.744991     192.168.2.44          202.117.1.13          TCP         54 61378 → 80 [ACK] Seq=1 Ack=1 Win=263424 Len=0
   241 7.745010     192.168.2.44          202.117.1.13          TCP         54 61379 → 80 [ACK] Seq=1 Ack=1 Win=263424 Len=0
   242 7.745153     192.168.2.44          202.117.1.13          HTTP      1189 GET / HTTP/1.1
   243 7.746951     202.117.1.13          192.168.2.44          TCP         54 80 → 61378 [ACK] Seq=1 Ack=1136 Win=64128 Len=0
   244 7.749095     202.117.1.13          192.168.2.44          TCP       8694 80 → 61378 [PSH, ACK] Seq=1 Ack=1136 Win=64128 Len=8640 [TCP PDU reassembled in 249]
   245 7.749125     192.168.2.44          202.117.1.13          TCP         54 61378 → 80 [ACK] Seq=1136 Ack=8641 Win=263424 Len=0
   246 7.749211     202.117.1.13          192.168.2.44          TCP       2934 80 → 61378 [ACK] Seq=8641 Ack=1136 Win=64128 Len=2880 [TCP PDU reassembled in 249]
   247 7.749233     192.168.2.44          202.117.1.13          TCP         54 61378 → 80 [ACK] Seq=1136 Ack=11521 Win=263424 Len=0
   248 7.749346     202.117.1.13          192.168.2.44          TCP       1494 80 → 61378 [ACK] Seq=11521 Ack=1136 Win=64128 Len=1440 [TCP PDU reassembled in 249]
   249 7.749920     202.117.1.13          192.168.2.44          HTTP      1298 HTTP/1.1 200 OK  (text/html)
   250 7.749947     192.168.2.44          202.117.1.13          TCP         54 61378 → 80 [ACK] Seq=1136 Ack=14205 Win=263424 Len=0
   251 7.770256     192.168.2.44          202.117.1.13          HTTP      1094 GET /style/xjnew611.css HTTP/1.1
   252 7.770369     192.168.2.44          202.117.1.13          HTTP      1077 GET /js/jquery.min.js HTTP/1.1
   253 7.770947     192.168.2.44          202.117.1.13          TCP         66 61380 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
   254 7.771146     192.168.2.44          202.117.1.13          TCP         66 61381 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
   255 7.771373     192.168.2.44          202.117.1.13          TCP         66 61382 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
   256 7.771540     192.168.2.44          202.117.1.13          TCP         66 61383 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
   257 7.772676     202.117.1.13          192.168.2.44          TCP       2934 80 → 61378 [ACK] Seq=14205 Ack=2176 Win=64128 Len=2880 [TCP PDU reassembled in 260]
   258 7.772707     192.168.2.44          202.117.1.13          TCP         54 61378 → 80 [ACK] Seq=2176 Ack=17085 Win=263424 Len=0
   259 7.773715     202.117.1.13          192.168.2.44          TCP       1494 80 → 61378 [ACK] Seq=17085 Ack=2176 Win=64128 Len=1440 [TCP PDU reassembled in 260]
   260 7.773715     202.117.1.13          192.168.2.44          HTTP       251 HTTP/1.1 200 OK  (text/css)
   261 7.773740     192.168.2.44          202.117.1.13          TCP         54 61378 → 80 [ACK] Seq=2176 Ack=18722 Win=263424 Len=0
   262 7.773968     192.168.2.44          202.117.1.13          HTTP      1101 GET /_sitegray/_sitegray_d.css HTTP/1.1
   263 7.774253     202.117.1.13          192.168.2.44          TCP         66 80 → 61380 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1440 SACK_PERM WS=128
   264 7.774253     202.117.1.13          192.168.2.44          TCP         66 80 → 61381 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1440 SACK_PERM WS=128
   265 7.774253     202.117.1.13          192.168.2.44          TCP         66 80 → 61382 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1440 SACK_PERM WS=128
   266 7.774253     202.117.1.13          192.168.2.44          TCP         66 80 → 61383 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1440 SACK_PERM WS=128
   267 7.774253     202.117.1.13          192.168.2.44          TCP         54 80 → 61379 [ACK] Seq=1 Ack=1024 Win=64128 Len=0
   268 7.774325     192.168.2.44          202.117.1.13          TCP         54 61380 → 80 [ACK] Seq=1 Ack=1 Win=263424 Len=0
   269 7.774338     192.168.2.44          202.117.1.13          TCP         54 61381 → 80 [ACK] Seq=1 Ack=1 Win=263424 Len=0
   270 7.774343     192.168.2.44          202.117.1.13          TCP         54 61382 → 80 [ACK] Seq=1 Ack=1 Win=263424 Len=0
   271 7.774356     192.168.2.44          202.117.1.13          TCP         54 61383 → 80 [ACK] Seq=1 Ack=1 Win=263424 Len=0
   272 7.774430     192.168.2.44          202.117.1.13          HTTP      1089 GET /index.vsb.css HTTP/1.1
```

图6 客户机与服务器的通信过程包（片段）

可以看到，在访问网站时，显示通信的端口为`80`，这是`HTTP`的默认服务端口。另外可以看到，出现了多个端口并行请求连接的情况，如上图中的`61378` `61379` `61380`等等。这是因为现代浏览器会**并行**加载网页中的资源，例如 **HTML**、**CSS**、**JavaScript** 文件、图片和其他内容。每个并行请求可能会创建一个新的 TCP 连接，并分配一个随机的源端口。下面我们会以`61378`端口为例来分析实验要求分析的各类报文。

工作基本流程总览

对等实体之间的通信是**逻辑上的**，也就是说并不是直接交换数据，而是经过**逐层封装/解封装**，由其下层传递而实现的。因此，`HTTP`工作时必须依靠下层的`TCP`来传输数据，也必须在`TCP`连接建立之后才能工作。以下给出了一个描述`C/S`间通信进程中协议的工作示意图：

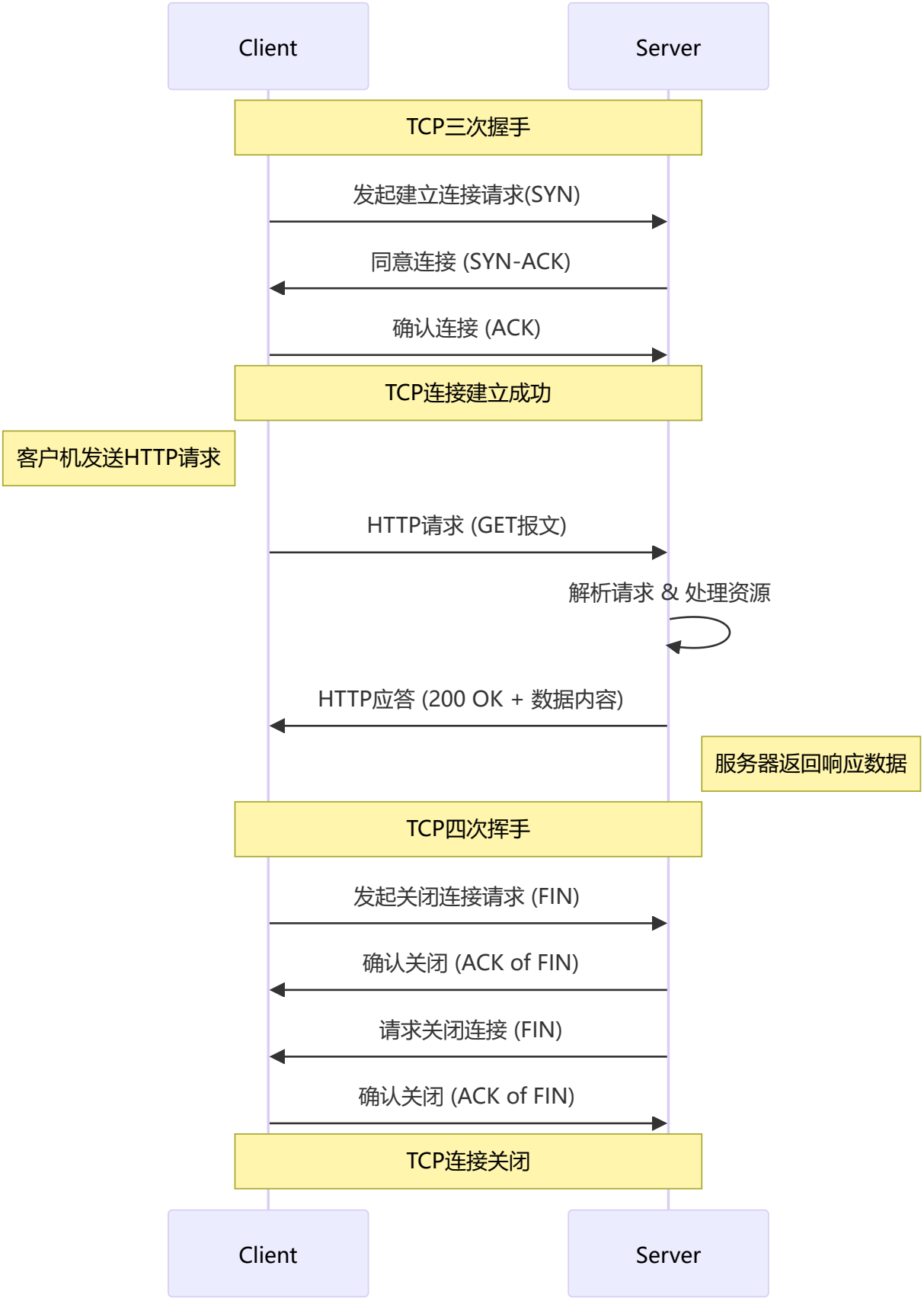
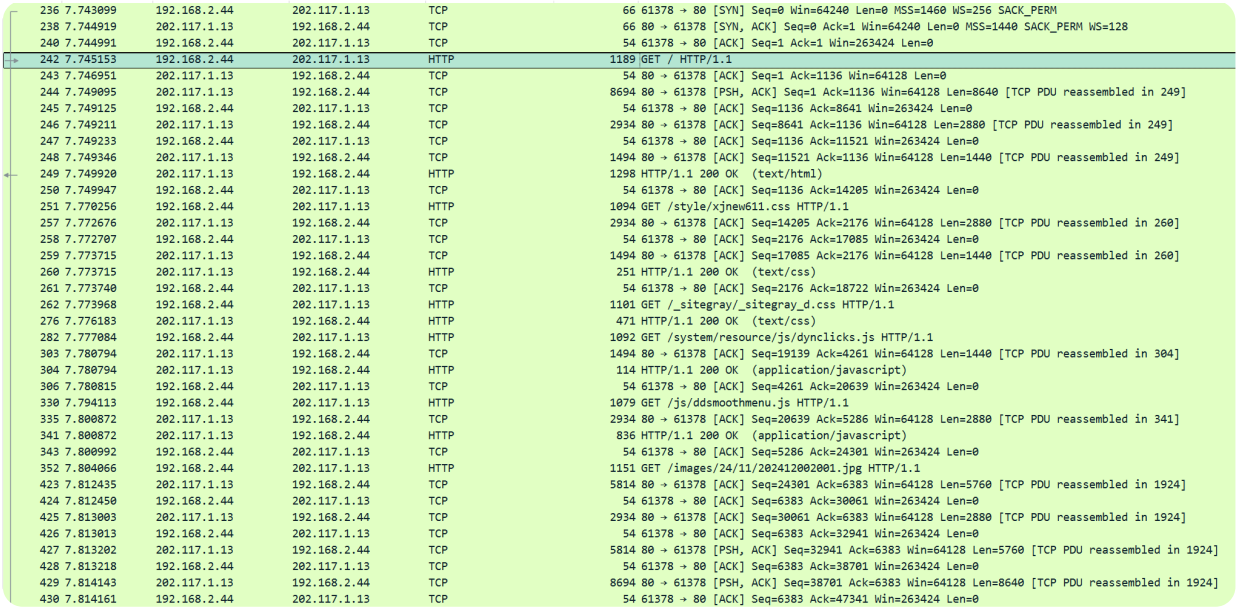


图7：HTTP与TCP协议工作流程

HTTP协议工作过程分析

如上文所说，以 `61378` 端口为例，来分析 `HTTP` 协议的工作过程。



236	7.743099	192.168.2.44	202.117.1.13	TCP	66 61378 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
238	7.744919	202.117.1.13	192.168.2.44	TCP	66 80 → 61378 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1440 SACK_PERM WS=128
240	7.744991	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=1 Ack=1 Win=263424 Len=0
242	7.745153	192.168.2.44	202.117.1.13	HTTP	1189 GET / HTTP/1.1
243	7.746951	202.117.1.13	192.168.2.44	TCP	54 80 → 61378 [ACK] Seq=1 Ack=1136 Win=64128 Len=0
244	7.749095	202.117.1.13	192.168.2.44	TCP	8694 80 → 61378 [PSH, ACK] Seq=1 Ack=1136 Win=64128 Len=8640 [TCP PDU reassembled in 249]
245	7.749125	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=1136 Ack=8641 Win=263424 Len=0
246	7.749211	202.117.1.13	192.168.2.44	TCP	2934 80 → 61378 [ACK] Seq=8641 Ack=1136 Win=64128 Len=2880 [TCP PDU reassembled in 249]
247	7.749233	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=1136 Ack=11521 Win=263424 Len=0
248	7.749346	202.117.1.13	192.168.2.44	TCP	1494 80 → 61378 [ACK] Seq=11521 Ack=1136 Win=64128 Len=1440 [TCP PDU reassembled in 249]
249	7.749920	202.117.1.13	192.168.2.44	HTTP	1298 HTTP/1.1 200 OK (text/html)
250	7.749947	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=1136 Ack=14205 Win=263424 Len=0
251	7.770256	192.168.2.44	202.117.1.13	HTTP	1094 GET /style/xjnew611.css HTTP/1.1
257	7.772676	202.117.1.13	192.168.2.44	TCP	2934 80 → 61378 [ACK] Seq=14205 Ack=2176 Win=64128 Len=2880 [TCP PDU reassembled in 260]
258	7.772707	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=2176 Ack=17085 Win=263424 Len=0
259	7.773715	202.117.1.13	192.168.2.44	TCP	1494 80 → 61378 [ACK] Seq=17085 Ack=2176 Win=64128 Len=1440 [TCP PDU reassembled in 260]
260	7.773715	202.117.1.13	192.168.2.44	HTTP	251 HTTP/1.1 200 OK (text/css)
261	7.773740	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=2176 Ack=18722 Win=263424 Len=0
262	7.773968	192.168.2.44	202.117.1.13	HTTP	1101 GET /_sitegray/_sitegray_d.css HTTP/1.1
276	7.776183	202.117.1.13	192.168.2.44	HTTP	471 HTTP/1.1 200 OK (text/css)
282	7.777084	192.168.2.44	202.117.1.13	HTTP	1092 GET /system/resource/js/dynclicks.js HTTP/1.1
303	7.780794	202.117.1.13	192.168.2.44	TCP	1494 80 → 61378 [ACK] Seq=19139 Ack=4261 Win=64128 Len=1440 [TCP PDU reassembled in 304]
304	7.780794	202.117.1.13	192.168.2.44	HTTP	114 HTTP/1.1 200 OK (application/javascript)
306	7.780815	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=4261 Ack=20639 Win=263424 Len=0
330	7.794113	192.168.2.44	202.117.1.13	HTTP	1079 GET /js/ddsmoothmenu.js HTTP/1.1
335	7.800872	202.117.1.13	192.168.2.44	TCP	2934 80 → 61378 [ACK] Seq=20639 Ack=5286 Win=64128 Len=2880 [TCP PDU reassembled in 341]
341	7.800872	202.117.1.13	192.168.2.44	HTTP	836 HTTP/1.1 200 OK (application/javascript)
343	7.800992	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=5286 Ack=24301 Win=263424 Len=0
352	7.804066	192.168.2.44	202.117.1.13	HTTP	1151 GET /images/24/11/202412002001.jpg HTTP/1.1
423	7.812435	202.117.1.13	192.168.2.44	TCP	5814 80 → 61378 [ACK] Seq=24301 Ack=6383 Win=64128 Len=5760 [TCP PDU reassembled in 1924]
424	7.812450	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=6383 Ack=30061 Win=263424 Len=0
425	7.813003	202.117.1.13	192.168.2.44	TCP	2934 80 → 61378 [ACK] Seq=30061 Ack=6383 Win=64128 Len=2880 [TCP PDU reassembled in 1924]
426	7.813013	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=6383 Ack=32941 Win=263424 Len=0
427	7.813202	202.117.1.13	192.168.2.44	TCP	5814 80 → 61378 [PSH, ACK] Seq=32941 Ack=6383 Win=64128 Len=5760 [TCP PDU reassembled in 1924]
428	7.813218	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=6383 Ack=38701 Win=263424 Len=0
429	7.814143	202.117.1.13	192.168.2.44	TCP	8694 80 → 61378 [PSH, ACK] Seq=38701 Ack=6383 Win=64128 Len=8640 [TCP PDU reassembled in 1924]
430	7.814161	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=6383 Ack=47341 Win=263424 Len=0

图8：截获部分有关61378端口的包

可以很清楚地看到，`242` 号包 `GET` 请求是在以上 `TCP` 三次握手之后才发送。下见该包中截获的报文细节：

```
Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    Host: www.xjtu.edu.cn\r\n
    Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/131.0.0.0 Safari/537.36\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7\r\n
    Accept-Encoding: gzip, deflate\r\n
    Accept-Language: zh-CN,zh;q=0.9\r\n
  > [...]Cookie: _ga_P7DT1QXXSK=GS1.1.1700662533.2.0.1700662533.0.0.0; _ga_7PM892EE02=GS1.1.1700662533.2.0.1700662533.60.0.0; _ga_ZYGQ8432T2=GS1.1.1700662533.2.0.1700662533.60.0.0; _ga=GA1.3.798618446.1700633427; tfstk=f5QoDZZ8Rg-Wd…
    \r\n
    [Response in frame: 249]
    [Full request URI: http://www.xjtu.edu.cn/]
```

图9：包242报文解析结果

请求行（Request Line）：

GET / HTTP/1.1：`GET` 为HTTP方法，表示客户端请求获取资源；其后表示获取内容的目录，此处 `/` 表示根路径；而后的 `HTTP/1.1` 表示使用的HTTP协议版本。

请求头（Request Headers）：

Host：表示目标服务器的域名；

Connection：`keep-alive` 表示客户端希望保持持久连接，复用 `TCP` 连接，这也是 `HTTP/1.1` 相比1.0能够支持的特性之一；

Upgrade-Insecure-Requests：`1` 表示客户端希望在可用的条件下升级为 `HTTPS` 连接；

User-Agent：UA字串的标准格式为 `浏览器标识（操作系统标识；加密等级标识；浏览器语言）渲染引擎标识 版本信息`。在浏览器发展早期，Netscape 的浏览器（代号 **Mozilla**）是第一个支持复杂网页功能的浏览器。很多网站当时根据 **User-Agent** 判断浏览器类型，只对 Mozilla 浏览器启用了高级功能。后来其他浏览器（包括 Internet Explorer、Chrome、Safari 等）为了兼容这些网站，

在 User-Agent 中伪装成 "Mozilla" ，确保网页功能正常运行。**AppleWebKit/537.36** 表示浏览器使用的是 **WebKit** 渲染引擎，最初用于 Safari 浏览器；**Chrome/131.0.0.0** 实际标识了使用的是 Google Chrome 浏览器；**Safari/537.36** 则因为出于兼容性目的，包含 Safari 的标识。

Accept：表示客户端可接受的内容类型。具体解析如下：

text/html：HTML 文本；

image/webp, image/apng：图片格式；

application/xml：XML 格式数据；

`*/*`：表示接受任何内容类型；

Accept-Encoding：表明客户端支持如图所示**两种内容压缩编码格式**；

Cookie：其头部携带了客户端保存的 Cookie 数据。`_ga` 系列通常是 Google Analytics 用于用户行为分析的标识符。

请求体（Request Body）：此处为 `GET` 报文，所以不附带本部分信息。

而相对应的响应报文 `249` 包的内容解析如下：

```
Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
    Date: Thu, 12 Dec 2024 17:27:22 GMT\r\n
    Server: **********\r\n
    X-Frame-Options: SAMEORIGIN\r\n
    Last-Modified: Thu, 12 Dec 2024 13:27:40 GMT\r\n
    Accept-Ranges: bytes\r\n
    Cache-Control: max-age=600\r\n
    Expires: Thu, 12 Dec 2024 17:37:22 GMT\r\n
    Vary: Accept-Encoding\r\n
    Content-Encoding: gzip\r\n
    ETag: "eaca-62912ada9cb00-gzip"\r\n
  v Content-Length: 13754\r\n
      [Content length: 13754]
    Keep-Alive: timeout=5, max=100\r\n
    Connection: Keep-Alive\r\n
    Content-Type: text/html\r\n
    Content-Language: zh-CN\r\n
    \r\n
    [Request in frame: 242]
    [Time since request: 0.004767000 seconds]
    [Request URI: /]
    [Full request URI: http://www.xjtu.edu.cn/]
    Content-encoded entity body (gzip): 13754 bytes -> 60106 bytes
    File Data: 60106 bytes
```

图10：包249报文解析结果

状态行（Status Line）：

HTTP/1.1 200 OK：

HTTP/1.1是HTTP 协议的版本。**200**是常用的表示请求成功的状态码，表示请求成功。**OK**是状态描述，表示一切正常，服务器成功处理了请求。

响应头（Response Headers）：

Date：服务器生成响应的时间，使用 GMT 格式。

Server：服务器软件信息（具体内容被隐藏）。

X-Frame-Options：`SAMEORIGIN`表示网页只能在同源的情况下被嵌入到`<iframe>`中，增强安全性，防止点击劫持攻击。

Last-Modified：资源的最后修改时间。

Accept-Ranges：允许客户端进行分段下载（按字节范围请求数据）。

Cache-Control：`max-age=600`表示客户端可缓存该资源，缓存时间为 **600 秒**（10 分钟）。

Expires：缓存过期时间，与`Cache-Control`对应。

Vary：表明响应内容根据客户端的`Accept-Encoding`头部进行变化（如 gzip 压缩）。

Content-Encoding：服务器对响应内容进行了 **gzip 压缩**，减少传输的数据量。

ETag：`"eaca-62912ad8b00c-gzip"`为资源的唯一标识符，帮助客户端判断资源是否发生改变（用于缓存验证）。

Content-Length：表示压缩后的响应内容长度为 **13754 字节**。

Keep-Alive：`timeout=5, max=100`。表示连接的**空闲超时时间**为 5 秒，**最大请求数**为 100。

Connection：`Connection: Keep-Alive`。表示服务器保持持久连接，允许复用当前连接。

Content-Type:`text/html`，表明响应内容的类型是 **HTML 文本**。

Content-Language：`Content-Language: zh-CN`，表示内容的语言是简体中文。

Content-encoded entity body (gzip)：原始的 HTML 文件经过 **gzip 压缩**，压缩后的长度为 **13754 字节**，解压后长度为 **60106 字节**。

响应体（Response Body）：具体有关网站内容的代码，此处不展开了。

所以我们可以据此概括出`HTTP/1.1`协议工作流程：

a. 打开浏览器输入网址访问时，浏览器作为客户端向 Web 服务器的 80 端口发送建立一个TCP连接的请求；

b. Web服务器确认连接后，客户端向服务器发送一个请求报文，包含**请求行**、**请求头**和**请求正文**；

c. 服务器返回一个响应报文，包含**状态行**、**响应头**和**响应体**；

d. 发送完响应，维持连接，直至达到**空闲超时时间**或达到**最大请求数**而终止。

TCP 协议工作过程分析

我们仍然以此端口为例，对截获各包中体现`TCP`协议工作特性的作重点分析。

（1）三次握手建立连接

首先还是截图整体的三次握手的过程：

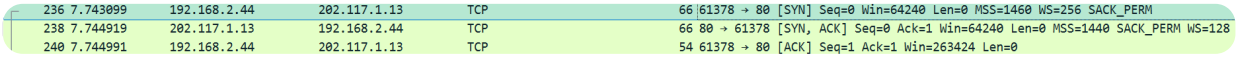


图11：TCP协议三次握手抓包

第一次握手：

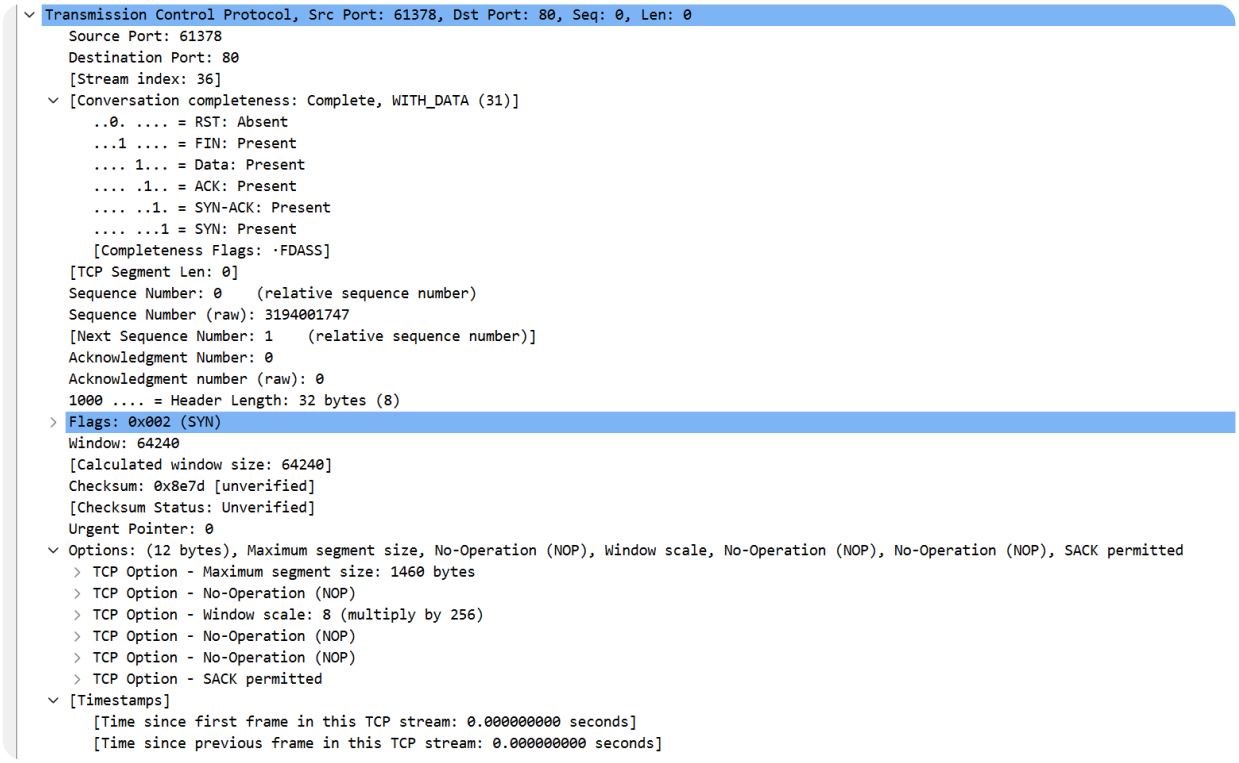


图12：TCP协议第一次握手

[Conversation completeness...]：判断对话完整度。其中**RST**是重置报文，在故障时设`1`。此处设`0`表示重置位缺省，连接正常；以下各位设`1`表示各个环节都正常进行。以上六个标志位首字母分别为`R F D A S S`，当对应位为`0`时用`·`表示缺省。

[TCP Segment Len: 0]：`TCP`报文**有效载荷**长度。此处为`0`，因为是`[SYN]`请求连接报文，所以不负载有效数据。

Sequence Number: 0：（相对）序列号为0。实际上在TCP 建立连接时，双方各自生成一个**随机的初始序列号（ISN, Initial Sequence Number）**，以防止连接被攻击或劫持。三次握手后双方的`ISN`会同步（*并非相同，而是同步，也就是同加减*）。在 Wireshark 中直接默认将收到序列号与`ISN`（设为`x`）作差得到相对序列号，便于识别TCP连接的进行标志；其下一行标注`(raw)`的即标识了客户端的真实`ISN`。

`1000 ....`：以二进制形式标出`TCP`头的长度。标准的`TCP`头长度是 20 Bytes，也就是 5字节，换算成二进制为`0101`。此有8字节，说明还有**额外的选项字段**，详见以下`Options`中列出的各种信息。

Flags：具体细节如下。

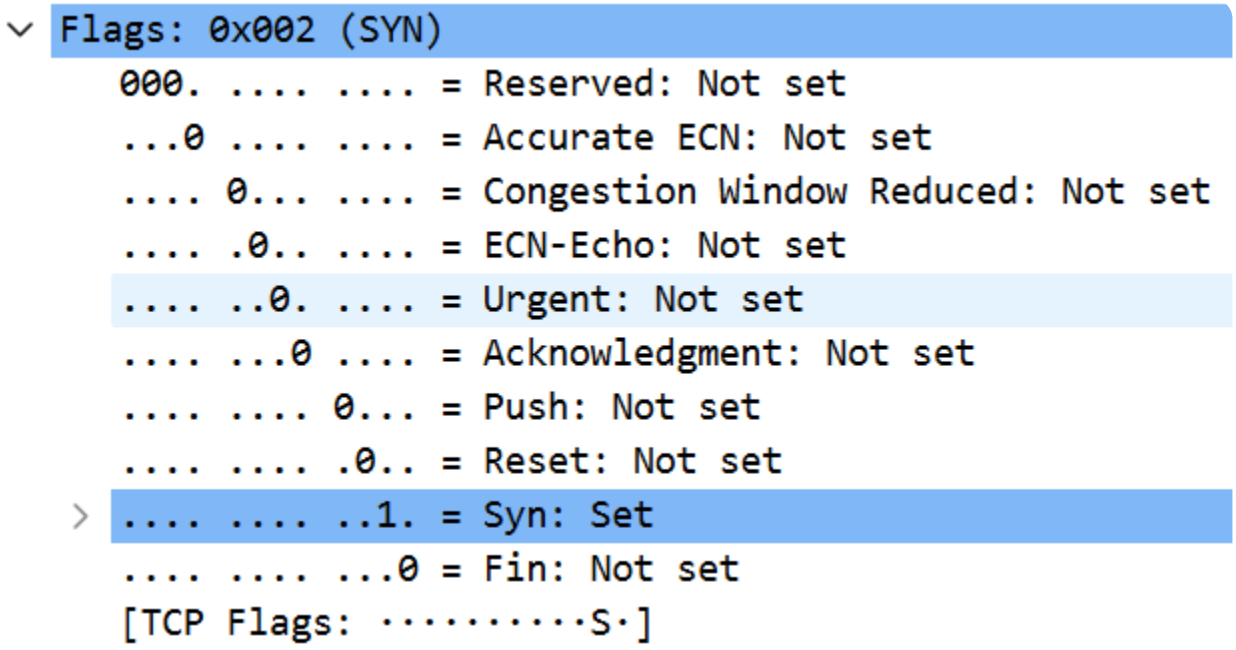


图13 Flags细节

只有`Syn`位置`1`，表明这是一个建立连接的`SYN`请求。

而后**Options**字段涵盖具体信息如下：

Maximum segment size（MSS）：协商最大TCP段长（不包括TCP头部）。此处$MSS = MTU - 20(IP头) - 20(TCP头) = 1460$，在**以太网**中$MTU$为**1500 字节**，因此有这一结果。在计算TCP头时不计选项字段长度，但要保证最终封装长度不超过MTU就可以。

Window：表明**发送方**的**接收**窗口大小，单位为字节。无论是服务器端还是用户机端，所发送的`Window`值都是**接收窗口大小**。

Checksum：校验和，是用于检测数据传输过程中错误的技术，主要以此来确保数据完整性。接收方接收到报文时会根据数据内容计算校验和，一致则说明数据无错。`[Unverified]`就表示还未经接收方校验。

Window Scale：通过缩放因子扩展窗口大小，允许更大的数据传输窗口。后值n代表放大倍数为2^n倍。在本情况中，也即**实际窗口大小**为$64240 \times 256 = 16,454,400$字节（约$16MB$）。

SACK 选项：启用**选择性（Selective）**确认，允许接收方告诉发送方它收到哪些不连续的数据块，提高传输效率，避免过多重传。

NOP：填充选项字段，确保长度对齐。

[Timestamps]：时间戳，以下两行分别表示距离**首帧**和**前一帧**的时间间隔，用于判断**TCP RTT（往返时间）测量**帮助发送方精确测量数据包的**往返时间（RTT）**或**防止序列号回绕问题**。

Urgent Pointer：置`0`表示不需要处理什么紧急数据。

下面展示后两次握手报文，对不特别之处不再额外说明：

图14 第二次握手

```
Transmission Control Protocol, Src Port: 61378, Dst Port: 80, Seq: 1, Ack: 1, Len: 0
   Source Port: 61378
   Destination Port: 80
   [Stream index: 36]
   [Conversation completeness: Complete, WITH_DATA (31)]
      ..0. .... = RST: Absent
      ...1 .... = FIN: Present
      .... 1... = Data: Present
      .... .1.. = ACK: Present
      .... ..1. = SYN-ACK: Present
      .... ...1 = SYN: Present
      [Completeness Flags: ·FDASS]
   [TCP Segment Len: 0]
   Sequence Number: 1    (relative sequence number)
   Sequence Number (raw): 3194001748
   [Next Sequence Number: 1    (relative sequence number)]
   Acknowledgment Number: 1    (relative ack number)
   Acknowledgment number (raw): 1283478329
   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: 1029
   [Calculated window size: 263424]
   [Window size scaling factor: 256]
   Checksum: 0x8e71 [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
   [Timestamps]
      [Time since first frame in this TCP stream: 0.001892000 seconds]
      [Time since previous frame in this TCP stream: 0.000072000 seconds]
   [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 238]
      [The RTT to ACK the segment was: 0.000072000 seconds]
      [iRTT: 0.001892000 seconds]
```

图15 第三次握手

第二次握手：

设服务端初始序号 **SEQ** 的值为 `y` 。确认接收到客户端的 **SYN** 报文，$ACK = 客户端的SEQ + 1$，表示确认收到客户机希望建立连接的请求，并期望收到序号为$SEQ + 1$的数据；**Flags** 中 **SYN** 与 **ACK** 都置 `1` ，表示同意建立连接，并请求客户机确认。**MSS** 变为 `1440` ，可能由于服务器有额外开销。

第三次握手：在第三次握手时，**SEQ**值+1，表明发送了第二条数据包；**ACK**值为窗口调整为 **1029**，这是基于**窗口缩放机制**，客户端根据当前缓冲区的实际可用空间得出的大小，经过窗口缩放后计算为 **263424** 字节。窗口缩放机制是一种常见的窗口调整机制，用于支持更大的窗口大小，以适应**高带宽延迟产品（BDP）网络**。

（2）四次挥手释放连接

按照传统的观点，释放连接的步骤如下：

客户端发送 FIN 包：当数据传输完成后，客户端会发送一个 **FIN（结束）** 标志的数据包，表示希望断开连接。

服务器回应 ACK 包：服务器收到 **FIN** 包后，会回应一个 **ACK** 包，确认关闭连接。

服务器发送 FIN 包：服务器也会发送一个 **FIN** 包，表示它准备关闭连接。

客户端回应 ACK 包：客户端收到服务器的 **FIN** 包后，发送一个 **ACK** 包，确认断开连接。至此，连接完全关闭。

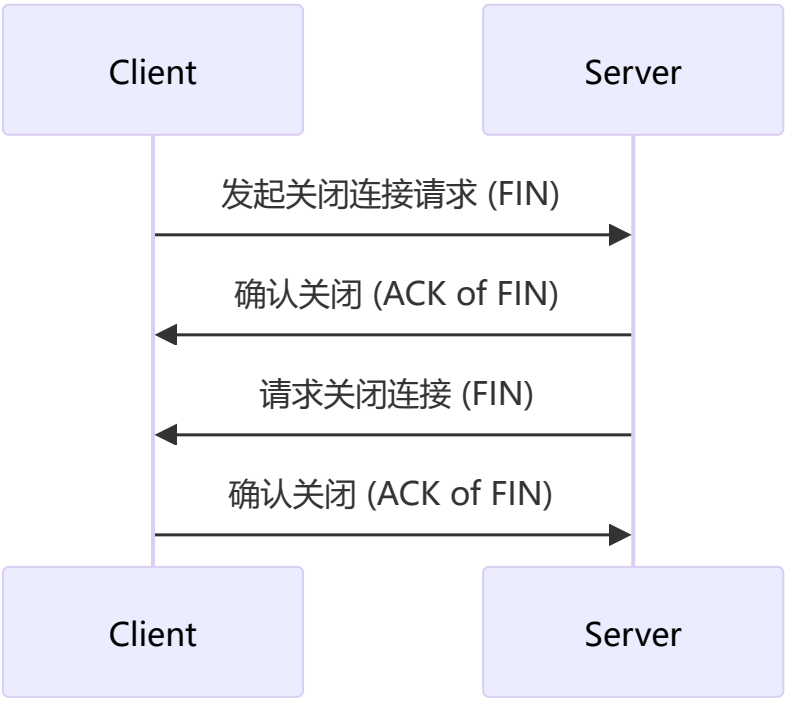


图16 TCP断开连接过程示意图

然而实际抓包结果有很大出入。

3138	8.138980	192.168.2.44	202.117.1.13	HTTP	1160 GET /img/fawu20230719.png HTTP/1.1
3171	8.144139	202.117.1.13	192.168.2.44	HTTP	872 HTTP/1.1 200 OK (PNG)
3194	8.198616	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=20787 Ack=652729 Win=263424 Len=0
3249	13.143661	202.117.1.13	192.168.2.44	TCP	54 80 → 61378 [FIN, ACK] Seq=652729 Ack=20787 Win=64128 Len=0
3251	13.143705	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [ACK] Seq=20787 Ack=652730 Win=263424 Len=0
3273	18.751335	192.168.2.44	202.117.1.13	TCP	54 61378 → 80 [FIN, ACK] Seq=20787 Ack=652730 Win=263424 Len=0
3280	18.753739	202.117.1.13	192.168.2.44	TCP	54 80 → 61378 [ACK] Seq=652730 Ack=20788 Win=64128 Len=0

图17 TCP断开连接实际抓包结果（最后几帧）

似乎和课本上的过程完全相反，而变成服务器先发送**FIN**请求了。这是为什么呢？

经过请教老师与同学，加上查阅相关资料，我或许知道了原因。这与之前所说**HTTP/1.1**支持的**Keep-Alive**机制下有关。让我们回顾一下之前分析的HTTP响应报文内容，注意到它对于**Keep-Alive**机制的维持时间有如此的规定：

> **Keep-Alive**：`timeout=5, max=100`。表示连接的空闲超时时间为 5 秒，最大请求数为 100。

也就是说，当到达了**5**秒的超时时间后，服务器会通过TCP协议发起**FIN**，这在抓包结果中体现为第`3249`号的报文（`[FIN, ACK]`）。它与`3194`号包相隔时间也恰恰接近于**5**秒，所以认为这是**持续连接**机制的超时断开条件触发的结果。我们可以通过检查其报文来验证猜想。

```
Transmission Control Protocol, Src Port: 80, Dst Port: 61378, Seq: 652729, Ack: 20787, Len: 0
    Source Port: 80
    Destination Port: 61378
    [Stream index: 36]
  > [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 652729    (relative sequence number)
    Sequence Number (raw): 1284131057
    [Next Sequence Number: 652730    (relative sequence number)]
    Acknowledgment Number: 20787    (relative ack number)
    Acknowledgment number (raw): 3194022534
    0101 .... = Header Length: 20 bytes (5)
  v Flags: 0x011 (FIN, 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
    > .... .... ...1 = Fin: Set
    v [TCP Flags: ·····A···F]
      v [Expert Info (Note/Sequence): This frame initiates the connection closing]
          [This frame initiates the connection closing]
          [Severity level: Note]
          [Group: Sequence]
    Window: 501
    [Calculated window size: 64128]
    [Window size scaling factor: 128]
    Checksum: 0xf111 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  v [Timestamps]
      [Time since first frame in this TCP stream: 5.400562000 seconds]
      [Time since previous frame in this TCP stream: 4.945045000 seconds]
```

```
Source Port: 61378
Destination Port: 80
[Stream index: 36]
> [Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 20787    (relative sequence number)
Sequence Number (raw): 3194022534
[Next Sequence Number: 20787    (relative sequence number)]
Acknowledgment Number: 652730    (relative ack number)
Acknowledgment number (raw): 1284131058
0101 .... = Header Length: 20 bytes (5)
v Flags: 0x010 (ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0... .... = Congestion Window Reduced: Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
    .... .... ...0 = Fin: Not set
    [TCP Flags: ·····A····]
Window: 1029
[Calculated window size: 263424]
[Window size scaling factor: 256]
Checksum: 0x8e71 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
v [Timestamps]
    [Time since first frame in this TCP stream: 5.400606000 seconds]
    [Time since previous frame in this TCP stream: 0.000044000 seconds]
v [SEQ/ACK analysis]
    [This is an ACK to the segment in frame: 3249]
    [The RTT to ACK the segment was: 0.000044000 seconds]
    [iRTT: 0.001892000 seconds]
```

```
Source Port: 61378
Destination Port: 80
[Stream index: 36]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 20787    (relative sequence number)
Sequence Number (raw): 3194022534
[Next Sequence Number: 20788    (relative sequence number)]
Acknowledgment Number: 652730    (relative ack number)
Acknowledgment number (raw): 1284131058
0101 .... = Header Length: 20 bytes (5)
Flags: 0x011 (FIN, 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
   .... .... ...1 = Fin: Set
      [Expert Info (Chat/Sequence): Connection finish (FIN)]
         [Connection finish (FIN)]
         [Severity level: Chat]
         [Group: Sequence]
   [TCP Flags: ·····A···F]
      [Expert Info (Note/Sequence): This frame undergoes the connection closing]
         [This frame undergoes the connection closing]
         [Severity level: Note]
         [Group: Sequence]
Window: 1029
[Calculated window size: 263424]
[Window size scaling factor: 256]
Checksum: 0x8e71 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
[Timestamps]
   [Time since first frame in this TCP stream: 11.008236000 seconds]
   [Time since previous frame in this TCP stream: 5.607630000 seconds]
```

```
Transmission Control Protocol, Src Port: 80, Dst Port: 61378, Seq: 652730, Ack: 20788,
   Source Port: 80
   Destination Port: 61378
   [Stream index: 36]
   [Conversation completeness: Complete, WITH_DATA (31)]
   [TCP Segment Len: 0]
   Sequence Number: 652730    (relative sequence number)
   Sequence Number (raw): 1284131058
   [Next Sequence Number: 652730    (relative sequence number)]
   Acknowledgment Number: 20788    (relative ack number)
   Acknowledgment number (raw): 3194022535
   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: 501
   [Calculated window size: 64128]
   [Window size scaling factor: 128]
   Checksum: 0xf110 [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
   [Timestamps]
      [Time since first frame in this TCP stream: 11.010640000 seconds]
      [Time since previous frame in this TCP stream: 0.002404000 seconds]
   [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 3273]
      [The RTT to ACK the segment was: 0.002404000 seconds]
      [iRTT: 0.001892000 seconds]
```

图18 四次挥手报文细节

可以看到，在第一个报文的`[Expert Info...]`中，注明了这是一个连接终止过程的开始帧（`This frame initiates the connection closing`），这是`Wireshark`解析器分析得出的结果。而后续的过程就与书中描述类似了。经过调整，我们画出抓包结果中的挥手过程示意图：

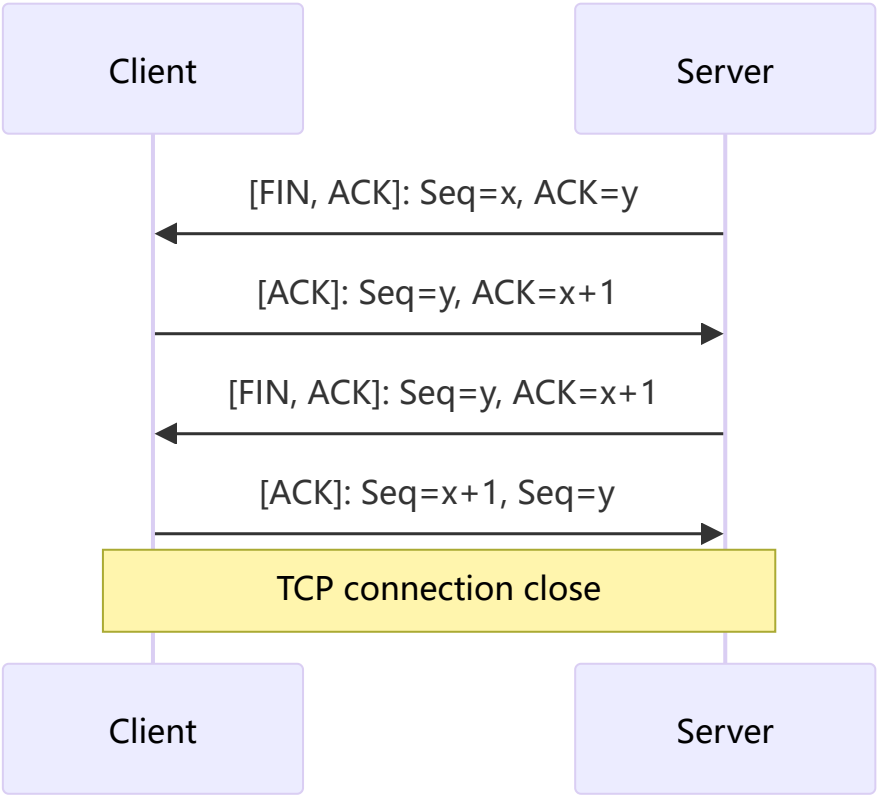


图19 实际连接中止流程图

六、实验总结与感想

几天下来，抓包过程似乎颇为简单，但是解析报文和撰写报告实在颇耗费气力，但也收获颇丰。我此前一直觉得这门课程有点抽象，所讲述的理论仿佛凭空而来，理解起来也很是费劲。此次实验真的深入了网络通信的流程之中，通过分析各个数据包，我对于 `HTTP` 和 `TCP` 协议的工作流程有了很深的体会，也很好地巩固了我在这方面的基础知识；同时，我也学到了很多课本上并未提及的拓展知识，比如 `HTTP` 协议与 `TCP` 协议的更多细节， `HTTP/1.1` 和 `HTTP/2` 等协议在 `1.0` 的基础上的改进，尤其是超时后由服务器发起 `FIN` 请求的四次挥手过程，这都令我大开眼界。另外，通过分析，我对这一流程有关的 `DNS` 的工作细节也有了初步的了解。此外，之后如果有时间， `TLSv1.3` 协议的一些细节也是值得探究的。 `HTTPS` 即是在 `HTTP` 的基础上使用 `TLS` 进行加密，默认端口也从 `80` 变成 `443` 。其中还有很多加密细节，留待此后钻研。

另：这篇报告极大锻炼了我的 `markdown` 语法，某种程度上也算一种收获（笑）。