《计算机网络》作业 1

1900017812 高乐耘*

2023 年 3 月 3 日

I. 实验环境

- 操作系统：Ubuntu 22.04.2 LTS (GNU/Linux 5.19.0-32-generic x86_64)
- 网卡：Intel Wireless-AC 9260
- 网络连接：PKU 5.2 GHz 无线校园网
- 浏览器：Mozilla Firefox 110.0
- Wireshark：Wireshark 3.6.2 (Git v3.6.2 packaged as 3.6.2-2)

II. 执行抓包

以网站 `https://www.baidu.com` 为例，尝试抓包。使用 `ping` 获取主机 `www.baidu.com` 的 IP 地址，结果显示：

```
$ ping www.baidu.com
PING www.a.shifen.com (182.61.200.6) 56(84) bytes of data.
64 bytes from 182.61.200.6 (182.61.200.6): icmp_seq=1 ttl=50 time=4.51 ms
64 bytes from 182.61.200.6 (182.61.200.6): icmp_seq=2 ttl=50 time=5.89 ms
64 bytes from 182.61.200.6 (182.61.200.6): icmp_seq=3 ttl=50 time=6.07 ms
64 bytes from 182.61.200.6 (182.61.200.6): icmp_seq=4 ttl=50 time=4.88 ms
```

由此得到目标主机的 IPv4 地址为 182.61.200.6。由于 HTTP 协议建立在 TCP/IP 协议之上，我们在抓包时可以过滤掉所有的非 TCP 分组。这样就得到了如下的捕获过滤器[1]：

```
tcp and host 182.61.200.6
```

关闭其它互联网应用，打开火狐浏览器，进入目标网站后，在 Wireshark 中选择当前接入 WAN 的网卡设备 wlp1s0（其物理设备为 Intel Wireless-AC 9260），指定捕获过滤器后，点击开始抓包按钮，转到浏览器目标网站页面，按 F12 唤出开发者页面，按 Ctrl+F5 进行硬刷新，此时浏览器丢弃所有页面缓存请求网站 HTML 页面及其链接的所有资源，如图 1。等待所有资源加载完毕后，回到 Wireshark 及时停止抓包。

*电子邮件地址: seeson@pku.edu.cn　手机号: 13759115414

[1]对于此处的 HTTPS 连接，一个更强的限制是 port 443，此处不加可以验证除 443 端口外，HTTPS 连接没有使用别的端口。

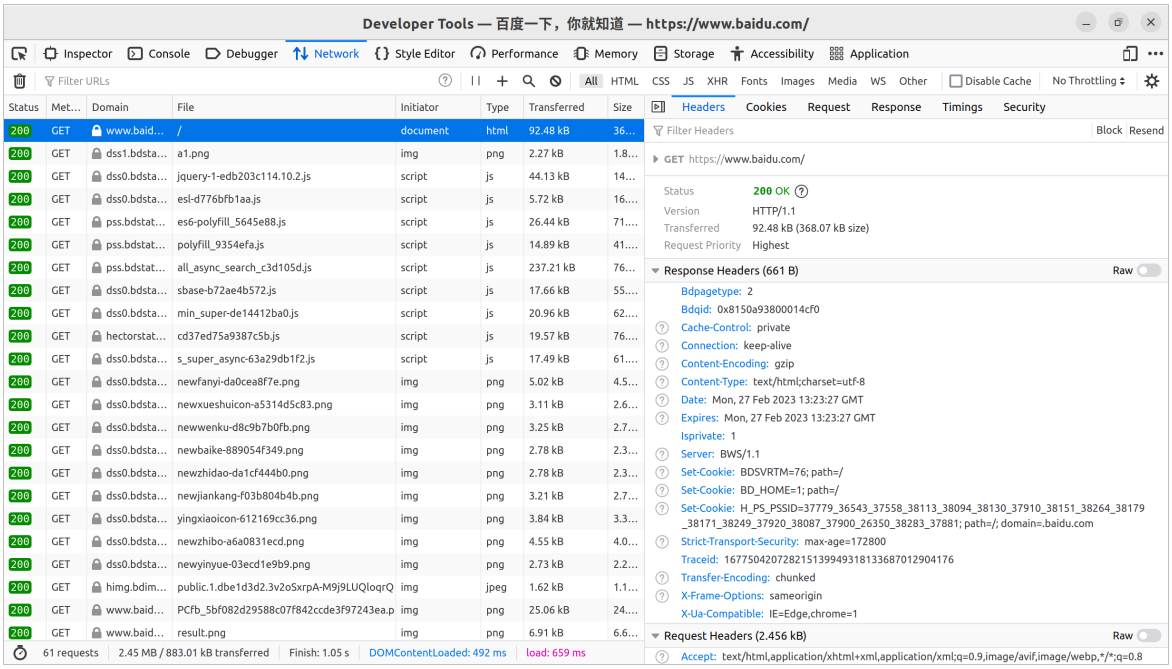

图 1: 浏览器 F12 工具捕捉的访问百度首页时的各项 HTTP 请求。共 61 个请求，由于页面所限没有完整显示。

III. 分析抓包结果

首先应用显示过滤器 tcp.port != 443，得到空结果；应用显示过滤器 tcp.port == 443，结果数与无显示过滤器一致。这说明本机与远程服务器的通信仅使用了 443 端口。

接着根据 HTTPS 连接建立的步骤，定位到第一个 Client Hello（第 21 个分组），它的前三个分组正好为 TCP 的三次握手，其后有多个 TLS 分组，应用层消息被加密，不可见。图 2 显示了这些分组。

现在来分析一下紧随其后的这个被加密的分组，即分组 26。从双方 IP 地址看出，这是一个由客户端发往服务器的分组。实际上，在 F12 工具面前，这个分组的内容很容易破解：因为我们使用了 Ctrl+F5 刷新网页，浏览器不能参照任何过去缓存的页面信息，因而无法并发或先行请求其它资源，故发出的第一个请求必定是对被访问页面 URL 的 GET 请求，那么这个分组必定属于该请求的一部分，请求内容如图 3。

图 3 显示，浏览器发出的首个 GET 请求的头部占据 2.456 kB，而在 Wireshark 的抓包结果（图 4）中，除去 TCP 及其它底层协议添加的头部共 0x36 字节后，剩下 2485 字节，大小与 2.456 kB 相当。从图 2 中还看出，自该分组（第 26 个分组）后服务器退回了大量响应分组，由此可以推断请求头后的空行必定在第 26 个分组中已经发出并被服务器接收，没有拆包现象发生，故这个分组的内容恰好就是浏览器发出的首个 GET 请求。如此我们便成功“破解”了 TLS 加密！

回到正题，现在我们得到了这个分组最上层——应用层的信息。同时图 4 还给出了其它的几层协议名称，物理层由于没有报头不在此列，加上我们破译出的应用层的 HTTP(S)，自底向上将它们列出就是：

- 物理层：无

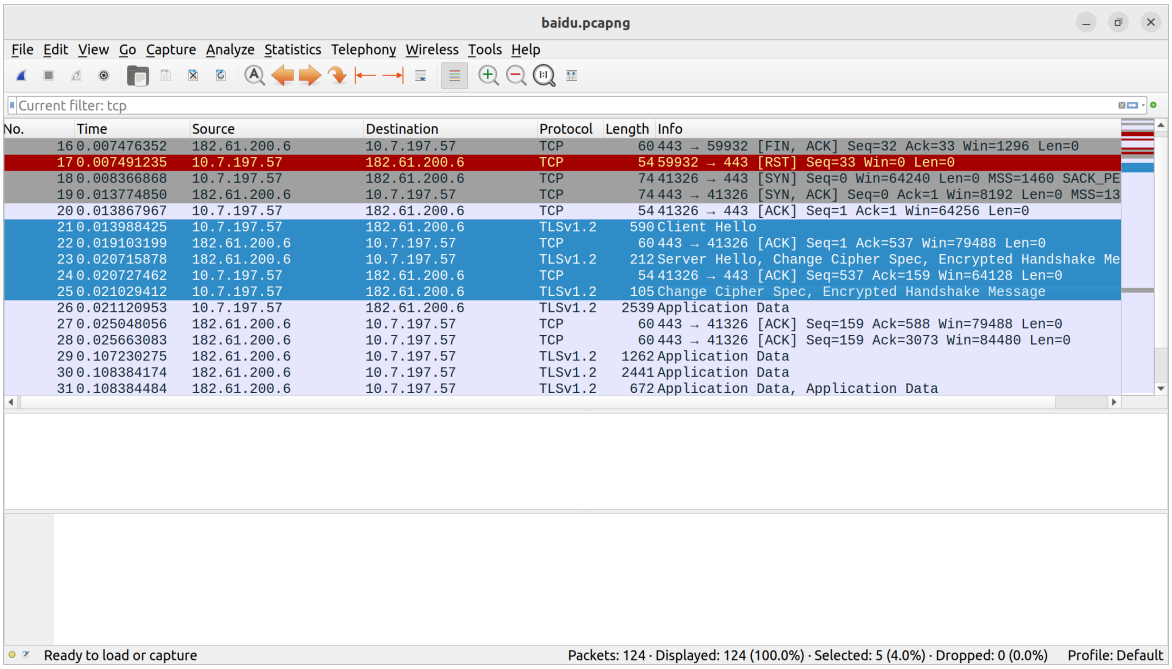

图 2: Wireshark 显示的客户端与服务器间的 TLS 握手过程，其前方为 TCP 的三次握手，后方有多个 TLS 分组。

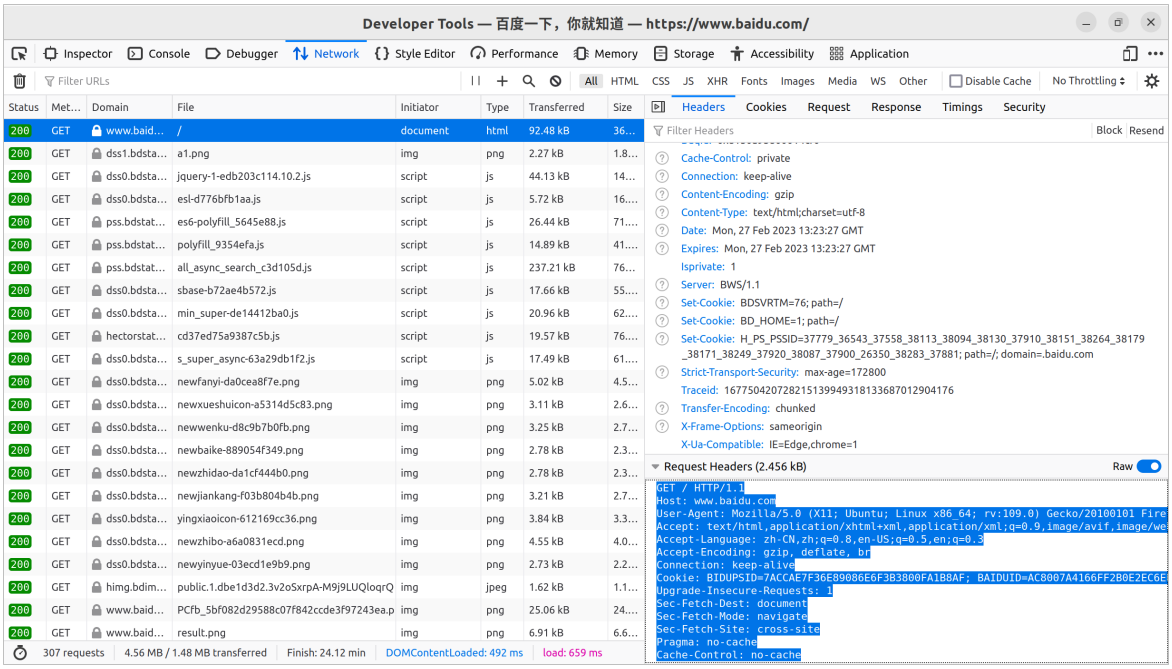

图 3: 浏览器 F12 工具显示的首个分组内容（右下角）。由于窗口大小所限，部分请求头没有完整显示。被选中的内容包括了这个 GET 请求的请求行和请求头，加上请求头后的空行（和空请求体）便是完整的应用层消息了。

- 链路层：以太网 II
- 网络层：IPv4
- 传输层：TCP 和 TLS

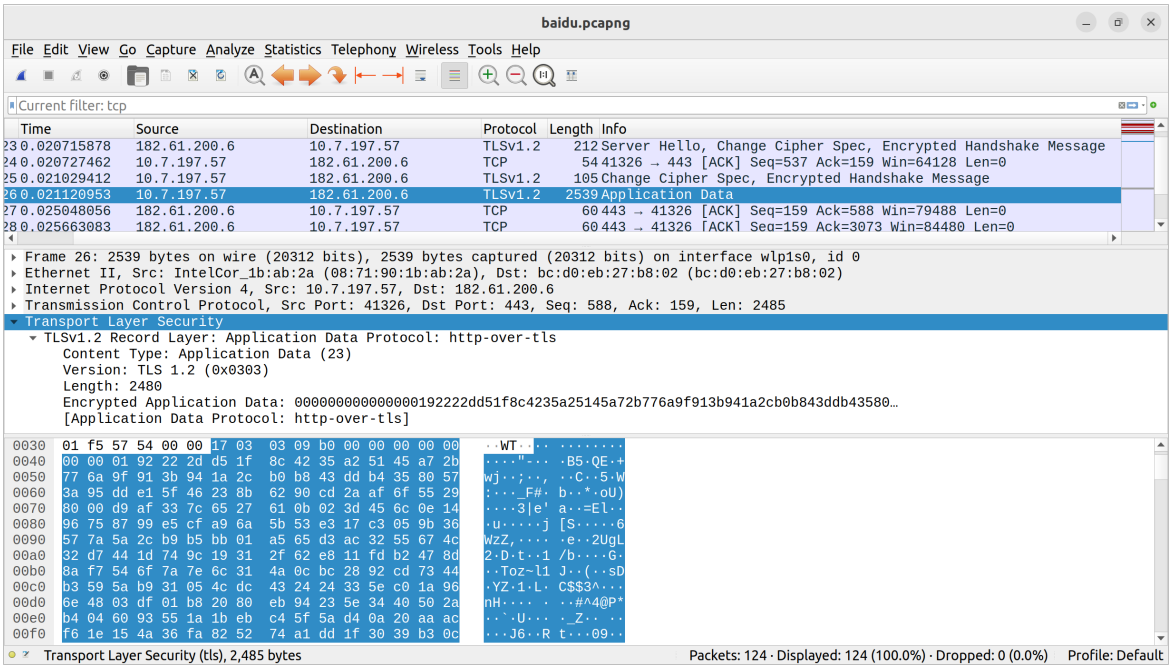

图 4: Wireshark 给出的首个 GET 请求内容详细信息。该部分内容被 TLS 加密难以破解。TLS 内容开始于地址 0x36，之前为 TCP 等协议添加的头部信息。

- 应用层：HTTP(S)

为了完成其它作业题目，下面让我们切换到网站 `http://vim.pku.edu.cn`。在此之前值得一提的是，我并未在 TLS 握手之后找到任何 Encrypted Alert 或置位 FIN 的 TCP 分组，但在 TLS 握手之前找到了它们（此处应为断开之前的 TLS 连接）。重新开启抓包刷新网页后等待片刻，我看到了带有 TCP Keep-Alive 信息的分组和对方的确认信息，这与图 1 中响应头的 Connection: keep-alive 对应，实际上这也是 HTTP/1.1 标准下的默认行为。

IV. 转向非强制 HTTPS 网站

这是我偶然发现的一个“落伍”的网站，没有实现强制从 HTTP 重定向到 HTTPS：

```
$ ping vim.pku.edu.cn
PING vim.pku.edu.cn(2001:da8:201:1920::731b:f038 (2001:da8:201:1920::731b:f038)) 56 data bytes
64 bytes from 2001:da8:201:1920::731b:f038 (2001:da8:201:1920::731b:f038): icmp_seq=1 ttl=59 time=4.22 ms
64 bytes from 2001:da8:201:1920::731b:f038 (2001:da8:201:1920::731b:f038): icmp_seq=2 ttl=59 time=3.30 ms
64 bytes from 2001:da8:201:1920::731b:f038 (2001:da8:201:1920::731b:f038): icmp_seq=3 ttl=59 time=4.10 ms
64 bytes from 2001:da8:201:1920::731b:f038 (2001:da8:201:1920::731b:f038): icmp_seq=4 ttl=59 time=4.81 ms
```

结合 ping 结果，这次我们使用这样的捕获过滤器，加上对 HTTP 80 端口的限制：

```
tcp and host 2001:da8:201:1920::731b:f038 and port 80
```

图 5 给出了浏览器发出的首个 GET 请求对应的分组，各层占用大小分别为：

- 链路层（以太网 II）头部：14 字节（占 2%）
- 网络层（IPv6）头部：40 字节（占 7%）
- 传输层（TCP）头部：32 字节（占 6%）

- 应用层（HTTP）消息：484 字节（占 85%）

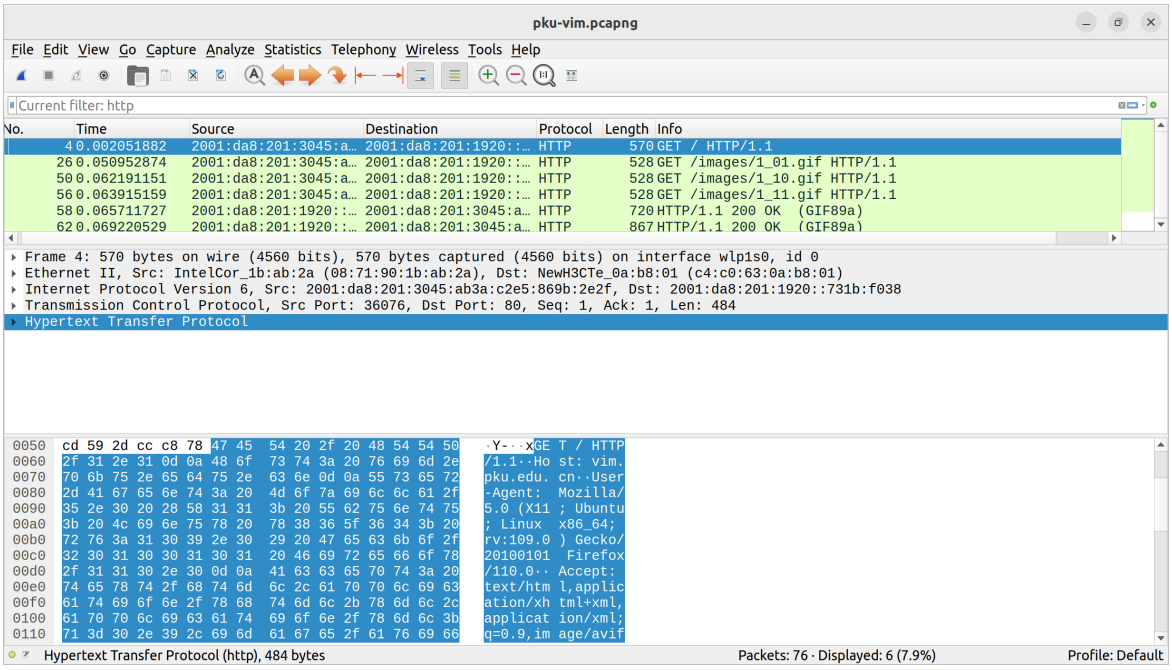

图 5: Wireshark 给出的首个 GET 请求内容详细信息。由于没有 TLS 加密，HTTP 报头被明文传输，可以直接解析。

这个 GET 请求的完整内容为：

```
GET / HTTP/1.1\r\n
Host: vim.pku.edu.cn\r\n
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/110.0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8\r\n
Accept-Language: zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3\r\n
Accept-Encoding: gzip, deflate\r\n
Connection: keep-alive\r\n
Upgrade-Insecure-Requests: 1\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache\r\n
\r\n
```

首行为请求行，包括请求的方法（GET），URL（/）和 HTTP 协议版本。Host 请求头是 HTTP/1.1 强制的。UA 请求头用于标识浏览器或其它用户代理。Accept* 请求头指定了浏览器期望的 MIME 类型，页面语言和内容编码方式。Connection: keep-alive 开启了 HTTP/1.1 长连接。Upgrade-Insecure-Requests 表示浏览器希望升级到更加安全的通信协议（如 HTTPS）。最后两行表示本次请求禁止使用缓存。

使用 F12 工具得到该请求的响应内容（因为 Wireshark 没有捕捉到）如下：

```
HTTP/1.1 200 OK\r\n
Connection: close\r\n
Content-Length: 9111\r\n
Accept-Ranges: bytes\r\n
Content-Type: text/html\r\n
Date: Fri, 03 Mar 2023 10:50:02 GMT\r\n
Etag: "2397-55e6405e7bf18"\r\n
Last-Modified: Mon, 20 Nov 2017 06:16:12 GMT\r\n
Server: Apache\r\n
```

X-Frame-Options: SAMEORIGIN\r\n
\r\n

首行为响应行，包括 HTTP 协议版本，状态码及其对应的简要的消息。Connection: close 在 HTTP/1.1 中被强制，如果不希望维持长连接。Content-Length 在 HTTP/1.1 中被强制，除非不使用长连接或使用 chunked 方式传输，故在此是不必要的。Accept-Ranges: bytes 允许用户代理进行断点续传。Content-Type 指定了内容的 MIME 类型。Date 指定了服务器当前的时间。Etag 用于用户代理在下次请求时回传以判断是否需要更新内容或使用缓存，Last-Modified 与之作用相同。Server 给出了服务器名称。X-Frame-Options 指定了在 iframe 中放置页面的要求，此处为同源要求，用于防止跨域攻击。

值得一提的是，该响应头 Content-Type 存在指定了文本类型的 MIME 而遗漏了字符集的缺陷，虽然在 HTML 网页中以 meta 的形式指定了编码为 utf-8，但足以造成比如 Python 中的 requests 等库的乱码错误（因为 requests 等不解析响应体）。

源和目标地址被编码在 IP 分组头部，各占用 16 字节（在 IPv4 中是 4 字节），如图 6。服务器的 IP 地址为 2001:da8:201:1920::731b:f038。

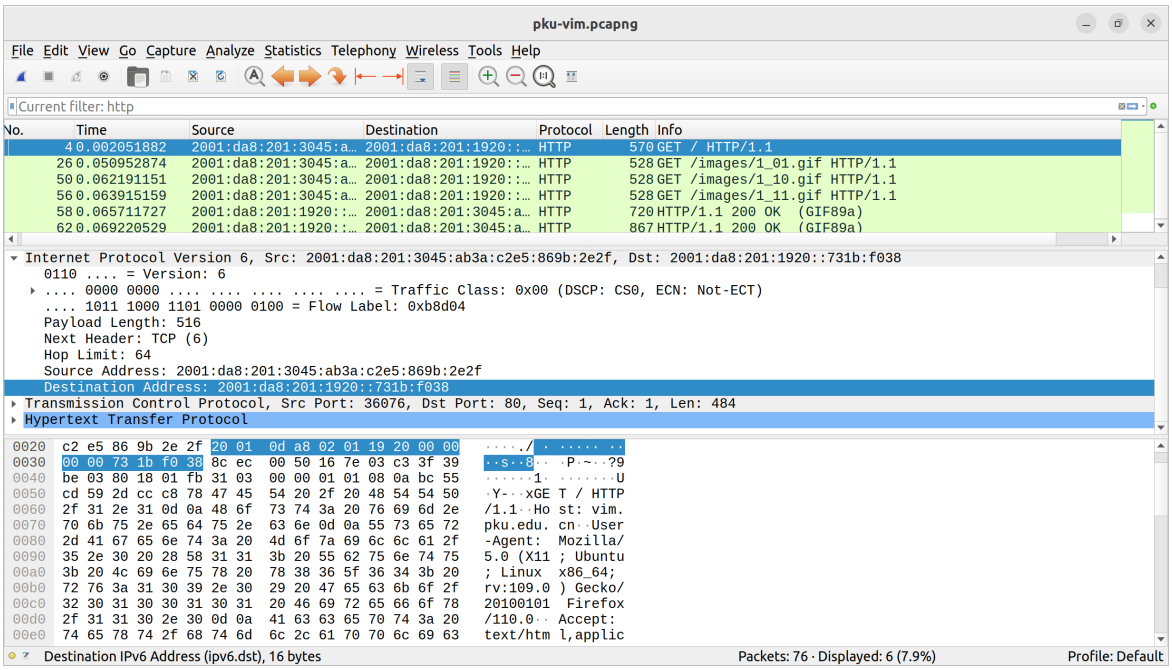

图 6: Wireshark 给出的 IP 分组头信息，此处为 IPv6 头，其中包含了两个各占 16 字节的源和目标地址。

应用显示过滤器 http，发现四个 GET 请求和两个响应，如图 7。两个丢失的响应可能源于图 8 中显示的错误。浏览器正确处理了这些错误，在 F12 工具中没有任何异常提示。

图 9 给出了本次抓包的流量图，仅保留了 HTTP 连接。注意到本机的端口号一直在变，检查服务器的响应头也发现了 Connection: close（确实这是一台不太先进的服务器）。图 10 中，我们还看到了第一次连接断开时的三次挥手（四次中的第二次和第三次合并了），因为收到客户端关闭连接的信息时服务器早已完成了所有数据的发送。

图 11 给出了客户端和服务器 HTTP 分组数目的统计结果，其中客户端发送 GET 请求被捕获到 4 个分组（占 67%），服务器退回 GET 响应被捕获到 2 个分组（占 33%）。由于至

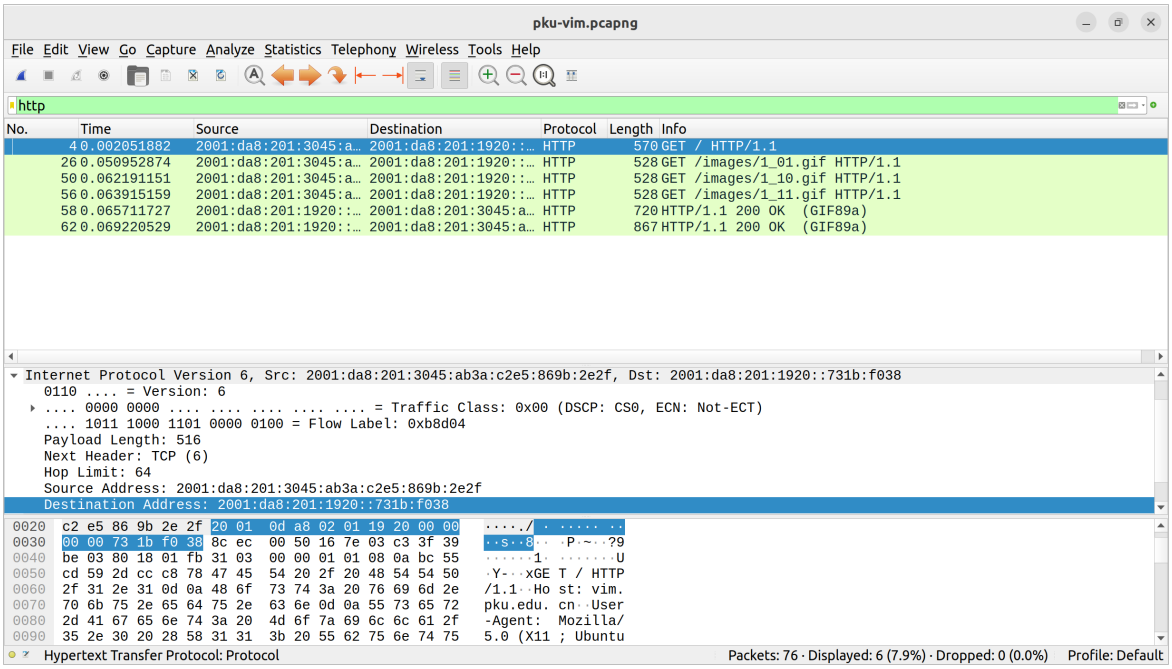

图 7: Wireshark 给出的 http 分组。图中包含四个 GET 请求和两个响应，另外两个 GET 请求的响应分组没有被正确捕捉。

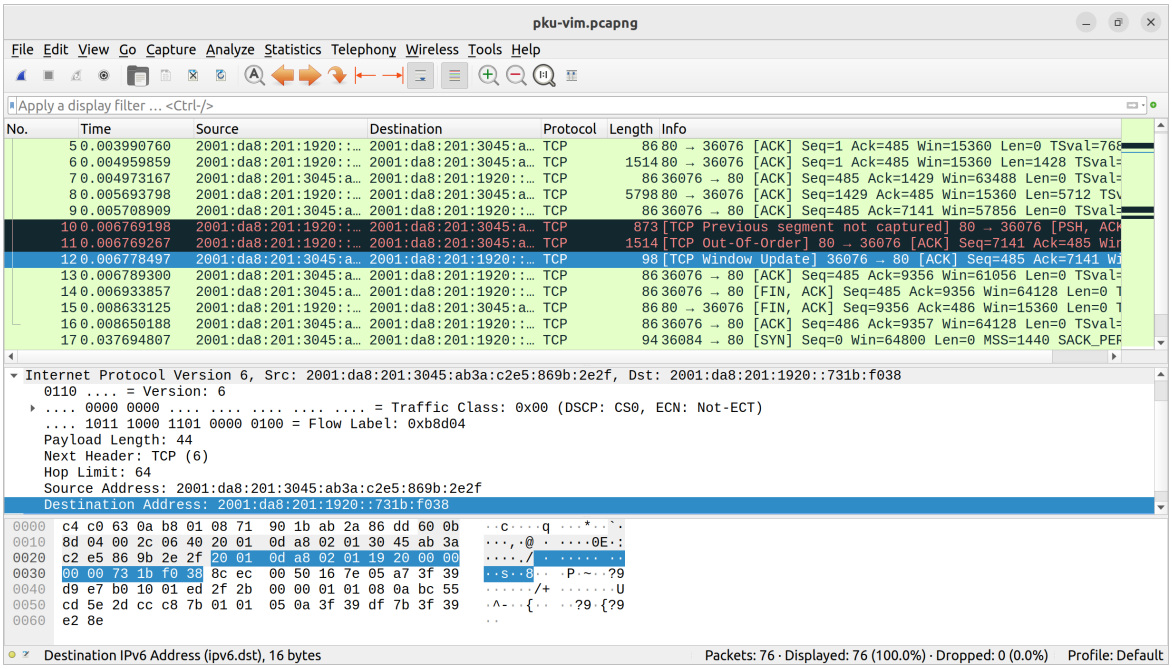

图 8: Wireshark 给出若干错误，这很可能对应着两个丢失的 GET 响应。两行黑色记录对应的错误在本次抓包中共出现两次。

少有两个响应没有被正常捕捉，这组数据是不可信的。实际情况下，HTTP 请求数与分组数没有确定的大小关系，因为较大的请求体或响应体可能会被拆包，由于网络中的各种原因也存在粘包现象，所以无法确定一个 HTTP 请求或响应将产生多少个分组。不过作为请求的发送者，我们应该能清楚地看到一个 GET 请求一般发出一个分组，而 POST 请求如果得到服务器的 100 响应需要发送请求体还将发出新的分组。

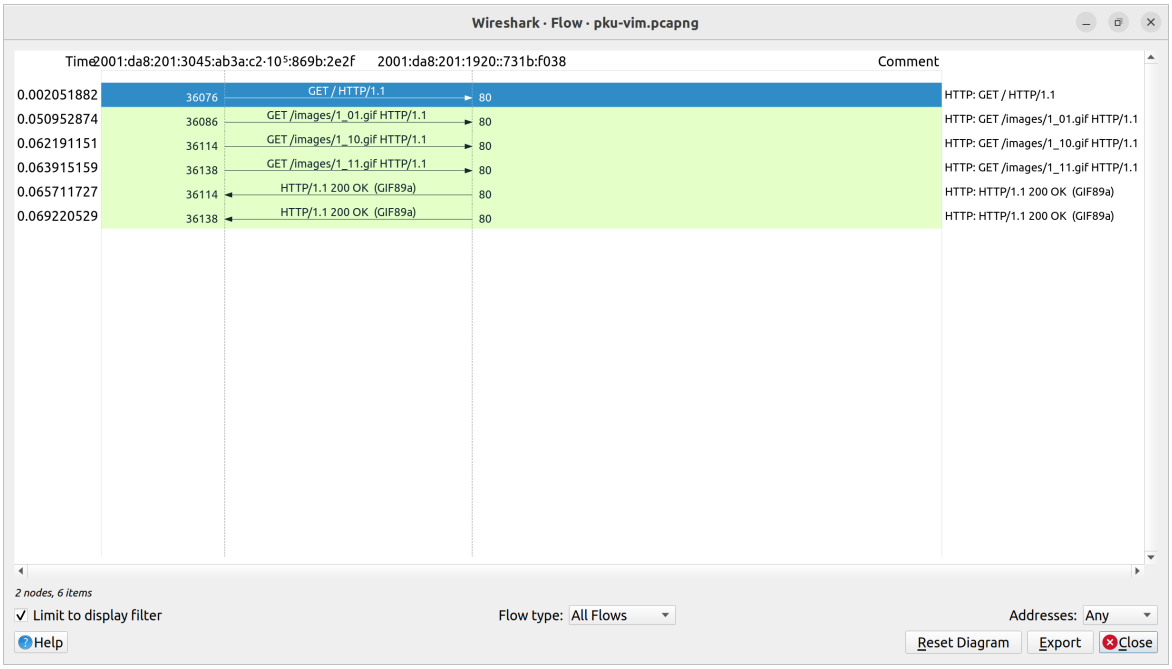

图 9: Wireshark 给出的流量图，仅保留了 HTTP 连接，其中本机端口号不断改变。有两个响应在抓包时丢失了，它们同样没有出现在上面的图中。

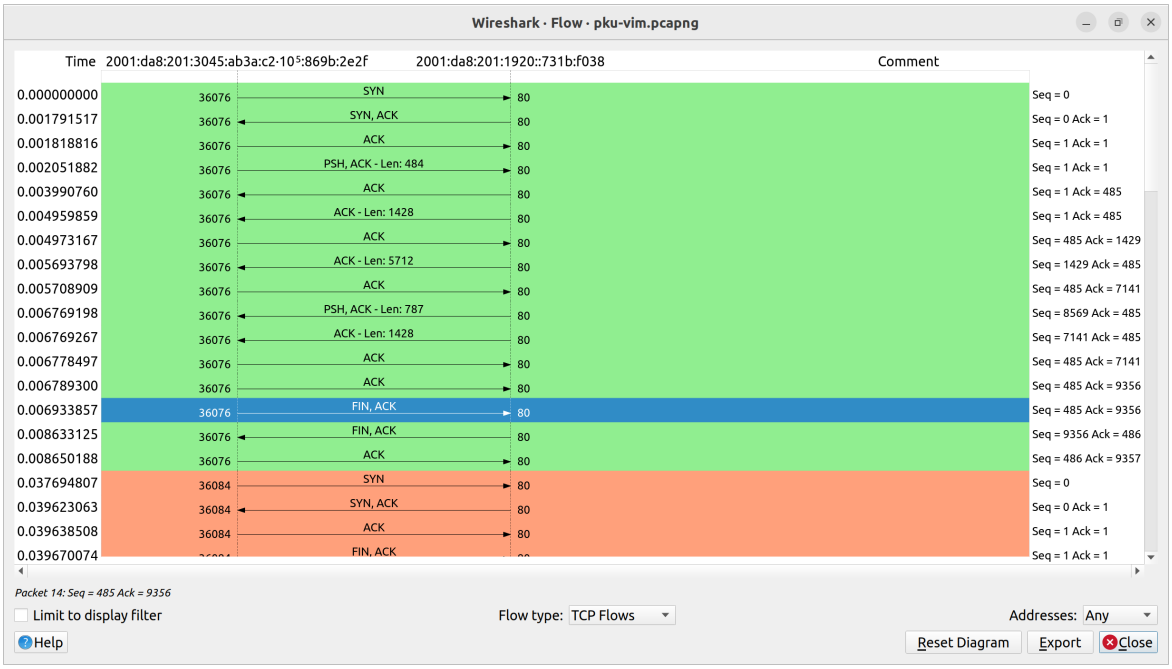

图 10: Wireshark 给出的流量图，仅保留了 TCP 连接，选中部分为第一次挥手，在本图中，总共只有三次挥手。

V. 更多的尝试

实际上，站点 `vim.pku.edu.cn` 的复杂性在于，与其连接可以混合使用 IPv4 和 IPv6，首先我们看看下面的域名解析结果。

```
$ nslookup vim.pku.edu.cn
```

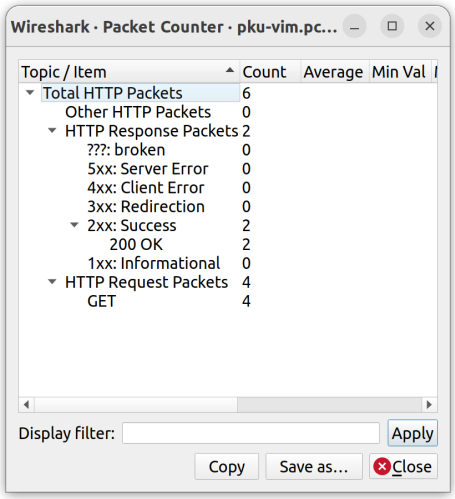

图 11: Wireshark HTTP 分组统计结果。必须指出，由于本次抓包有两个 HTTP 响应没有被正常捕捉，故该统计数据是不完整的。

```
Server:     127.0.0.53
Address:    127.0.0.53#53

Non-authoritative answer:
Name:   vim.pku.edu.cn
Address: 115.27.240.56
Name:   vim.pku.edu.cn
Address: 2001:da8:201:1920::731b:f038
```

我们可以如下修改捕获过滤器，来探究浏览器是否同时使用了 IPv4 和 IPv6 连接与服务器通信：

```
tcp and host vim.pku.edu.cn and port 80
```

如图 12，答案是肯定的。

图 13 给出了同时使用 IPv4 和 IPv6 连接的具体过程。本机先后向 IPv6 地址（分组 1）和 IPv4 地址（分组 2）发送了两个 SYN，但与 IPv4 地址的 TCP 连接请求先得到了服务器的回应（分组 3），此时本机立即回应了 ACK（分组 4），TCP 连接建立成功，浏览器随即发送了 GET 请求（分组 5）。此后本机虽然也与 IPv6 地址处的服务器（可能与 IPv4 地址对应的服务器相同也可能不相同）建立了 TCP 连接（分组 6 和 7），但随即将其断开（分组 8，14 和 15），并未传输任何数据。由此推测浏览器可能对同一域名解析的不同结果以非阻塞的方式同时尝试进行套接字连接，且只保留最先就绪的连接。

如果想要避免 IPv4 和 IPv6 同时连接，一个最直接的办法是使用 IP 地址访问网站。这并非总是有效，因为有的相同 IP 站点可能服务多个不同域名，但对于 `http://115.27.240.56` 而言，在浏览器地址栏中输入这个地址，可以正常打开。不幸的是，`http://2001:da8:201:1920::731b:f038` 并非有效的 URL，由于冒号在 URL 中的特殊地位。其正确形式为：`http://[2001:da8:201:1920::731b:f038]`。如果不知道这种方式，修改 hosts 也不失为一种有效的解决方案。当然，像 `https://115.27.240.56` 和 `https://[2001:da8:201:1920::731b:f038]` 这样的链接均是合法的，但它们一般都不能正常打开，因为 HTTPS 证书一般跟域名

而非 IP 地址绑定，对于这个网站同样如此。

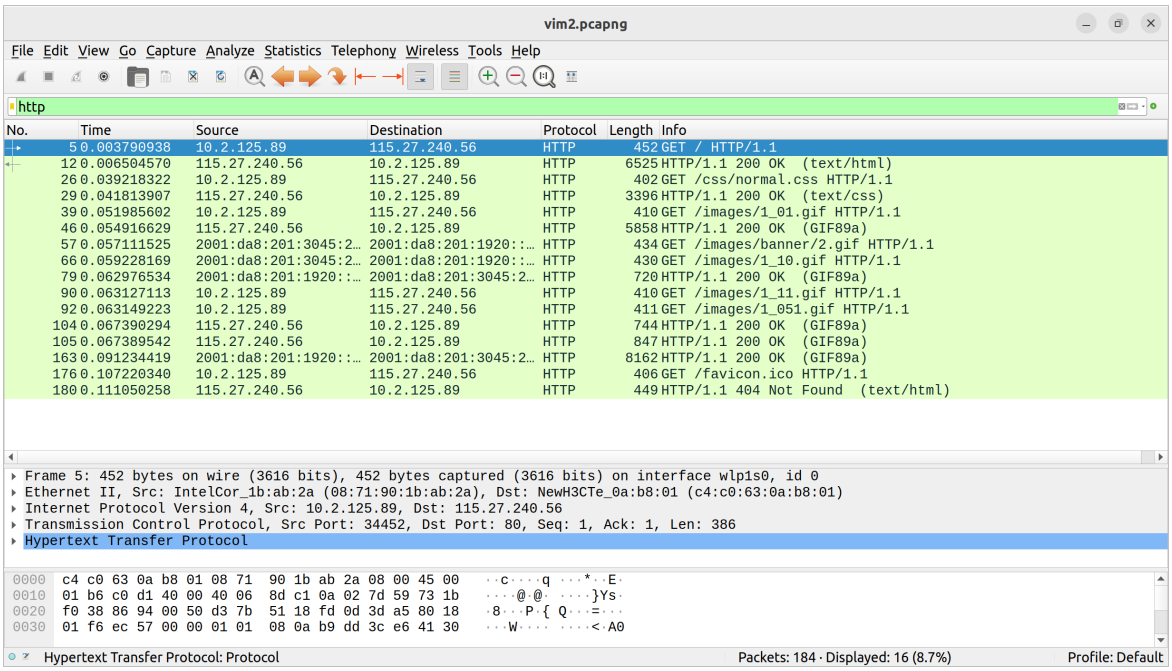

图 12: Wireshark HTTP 分组统计结果。注意到源和目的地址中不仅包括 IPv4 地址，还包括 IPv6 地址。

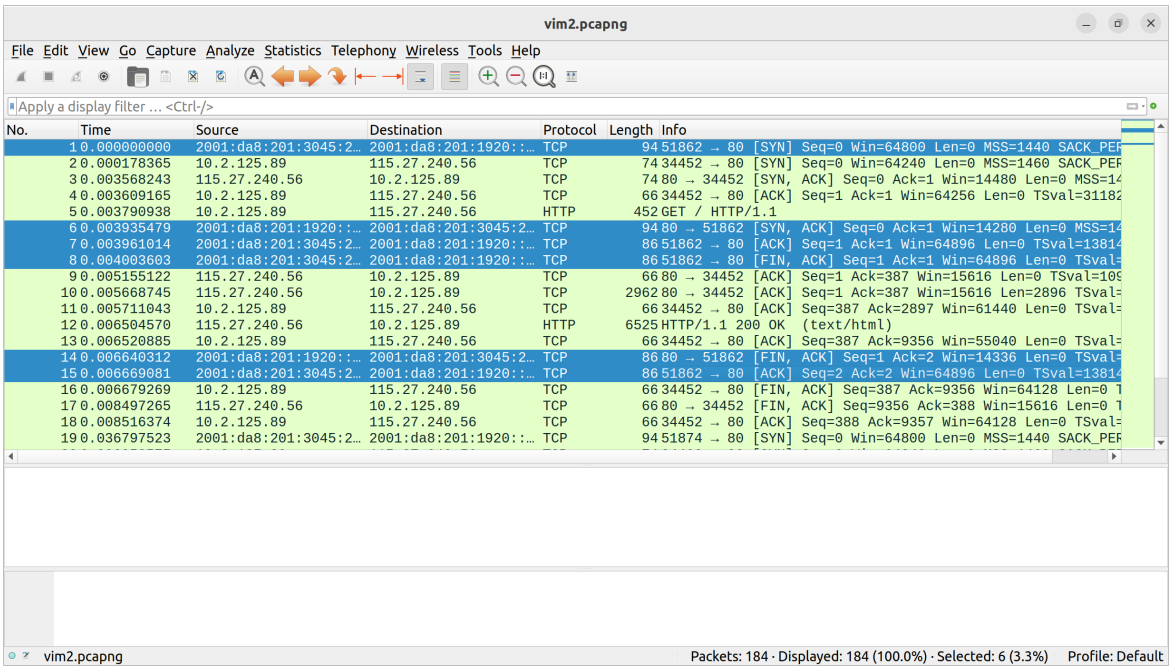

图 13: Wireshark 捕获的所有 TCP 连接。注意到选中部分对应本机与服务器的 TCP 握手和挥手过程，且握手与挥手之间没有任何数据传输。

图 12 中显示的 8 个 HTTP 请求与 F12 工具给出的完全一致，是多次重复实验偶然得到的结果。即使是同时考虑了 IPv4 和 IPv6 连接，也不能每次在 Wireshark 中看到所有的 HTTP 请求。但改用有线网卡进行实验，可以在大多数情况下看到所有请求，具体原因需要

学习后续课程之后仔细分析。

参考文献

[1] Kurose, J. F., Ross, K. W. (2021). Computer Networking: A Top-Down Approach. Boston, MA: Pearson. ISBN: 978-0-13-592861-5.