HTTP/3 LAB

Lab Report

Bar Binyamin Varsulker,

Ori Meir Kushnir,

Answers to Practical Questions

1. The file defines two server blocks — one for HTTPS (localhost:443) and one for HTTP (localhost:80). Each block tells Caddy what to do when a request comes in on that port.

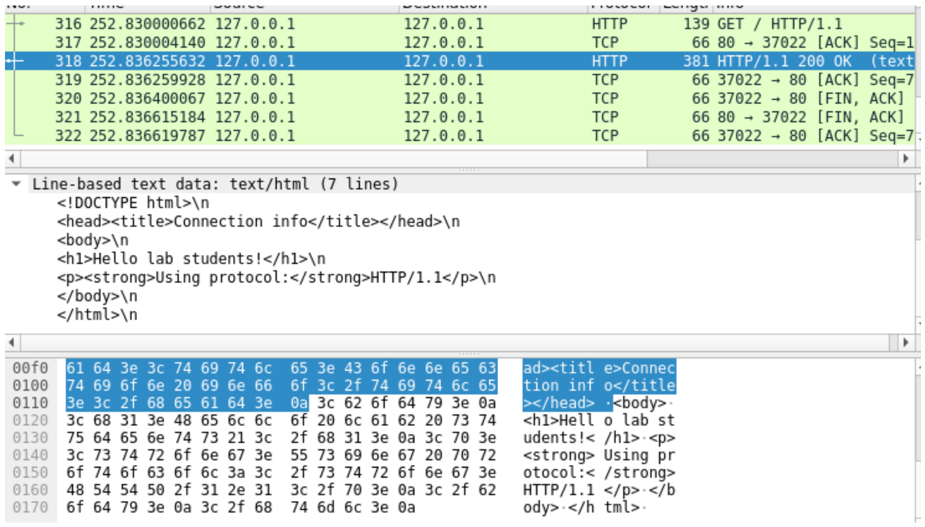
   Starting with localhost:443, this is the configuration for handling secure HTTPS requests. Inside the block, we first set the root directory for the server to /srv, meaning all requests will look for files starting from that directory. Then, file_server tells Caddy to serve static files (like HTML, CSS, images, etc.) from that root folder. The templates directive allows simple templating, so if you're using placeholders like {{.Host}} in your HTML files, Caddy can process them. Finally, tls internal tells Caddy to automatically generate and use a TLS certificate from its internal CA, which is super helpful for local development — you get HTTPS without needing to get a real certificate.

   The second block, localhost:80, is for regular HTTP traffic. It's basically doing the same as the first block — setting the root to /srv, enabling static file serving, and allowing template processing — but it doesn't include TLS because it's HTTP, not HTTPS.
   This kind of setup is useful for running a basic secure web server locally, where HTTPS is supported on port 443 with automatic internal certificates, and HTTP works on port 80 for fallback or redirecting purposes.

2. This command runs a Caddy server inside a Docker container in detached mode, meaning it runs in the background.
   The -p flags map ports from your host machine to the container. Port 80 is for HTTP, and port 443 is for HTTPS — it's mapped both for TCP (default) and explicitly for UDP as well. That UDP mapping is important because HTTP/3 (which Caddy supports) runs over QUIC, and QUIC uses UDP on port 443.
   The -v flags mount volumes. The first one, ~/site:/srv, maps your local site folder into the container's /srv directory — that's where Caddy will find your website files and your Caddyfile configuration. The second volume, caddy_data:/data, is where Caddy stores things like TLS certificates and internal state, so they persist even if the container is restarted.
   At the end, the command tells Docker to run the official Caddy image (caddy:latest) and start the server using caddy run, while explicitly pointing it to the configuration file located at /srv/Caddyfile, which is the one you mounted earlier.

3. curl (Client URL) is a CLI utility used for transferring data to or from servers using a variety of network protocols, including HTTP, HTTPS, FTP, SMTP, and others. It's commonly used by developers and system admins for downloading files, interacting with web APIs, debugging network connections, or automating web-related tasks. curl operates through simple commands entered directly into the terminal, allowing users to easily specify details like URLs, request methods, headers, and data payloads.

4.



```
316 252.830000662 127.0.0.1    127.0.0.1    HTTP   139 GET / HTTP/1.1
317 252.830004140 127.0.0.1    127.0.0.1    TCP     66 80 → 37022 [ACK] Seq=1
318 252.836255632 127.0.0.1    127.0.0.1    HTTP   381 HTTP/1.1 200 OK  (text
319 252.836259928 127.0.0.1    127.0.0.1    TCP     66 37022 → 80 [ACK] Seq=7
320 252.836400067 127.0.0.1    127.0.0.1    TCP     66 37022 → 80 [FIN, ACK]
321 252.836615184 127.0.0.1    127.0.0.1    TCP     66 80 → 37022 [FIN, ACK]
322 252.836619787 127.0.0.1    127.0.0.1    TCP     66 37022 → 80 [ACK] Seq=7
```

```
▾ Line-based text data: text/html (7 lines)
    <!DOCTYPE html>\n
    <head><title>Connection info</title></head>\n
    <body>\n
    <h1>Hello lab students!</h1>\n
    <p><strong>Using protocol:</strong>HTTP/1.1</p>\n
    </body>\n
    </html>\n
```

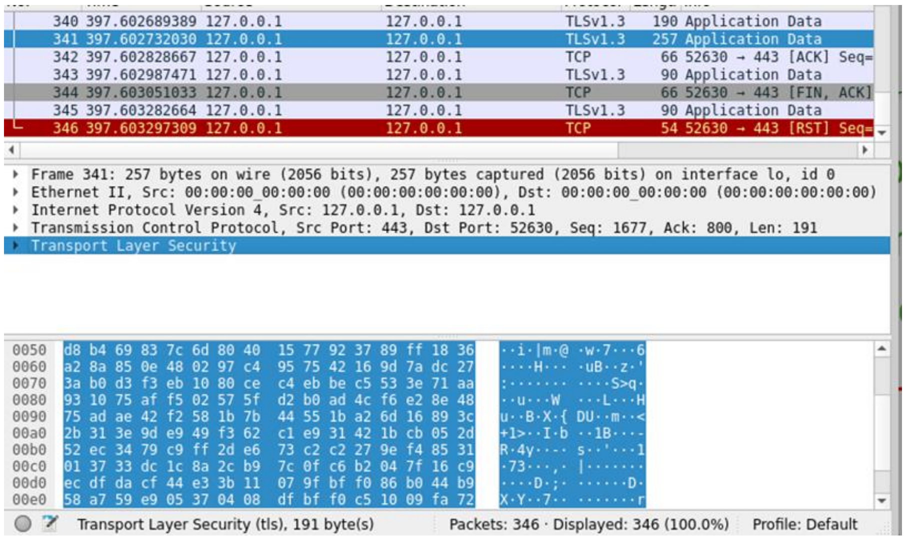
```
00f0  61 64 3e 3c 74 69 74 6c  65 3e 43 6f 6e 6e 65 63   ad><titl e>Connec
0100  74 69 6f 6e 20 69 6e 66  6f 3c 2f 74 69 74 6c 65   tion inf o</title
0110  3e 3c 2f 68 65 61 64 3e  0a 3c 62 6f 64 79 3e 0a   ></head> ·<body>·
0120  3c 68 31 3e 48 65 6c 6c  6f 20 6c 61 62 20 73 74   <h1>Hell o lab st
0130  75 64 65 6e 74 73 21 3c  2f 68 31 3e 0a 3c 70 3e   udents!< /h1>·<p>
0140  3c 73 74 72 6f 6e 67 3e  55 73 69 6e 67 20 70 72   <strong> Using pr
0150  6f 74 6f 63 6f 6c 3a 3c  2f 73 74 72 6f 6e 67 3e   otocol:< /strong>
0160  48 54 54 50 2f 31 2e 31  3c 2f 70 3e 0a 3c 2f 62   HTTP/1.1 </p>·</b
0170  6f 64 79 3e 0a 3c 2f 68  74 6d 6c 3e 0a            ody>·</h tml>·
```

When analyzing the HTTP/1.1 traffic in Wireshark packet captures, the most glaring disadvantage becomes immediately apparent: all data is transmitted in plain text. This means anyone with access to network traffic can easily read usernames, passwords, session cookies, and other sensitive information directly from the pcap file without any special tools or decryption keys.

This Wireshark capture perfectly demonstrates the security vulnerability of HTTP/1.1 I mentioned. The image shows:

- A packet capture with HTTP/1.1 traffic (packet #318 shows "HTTP/1.1 200 OK")
- The HTML content of the web page is fully visible in plain text.
- Both the ASCII representation (right side) and hex values (left side) of the data are completely readable.

5.

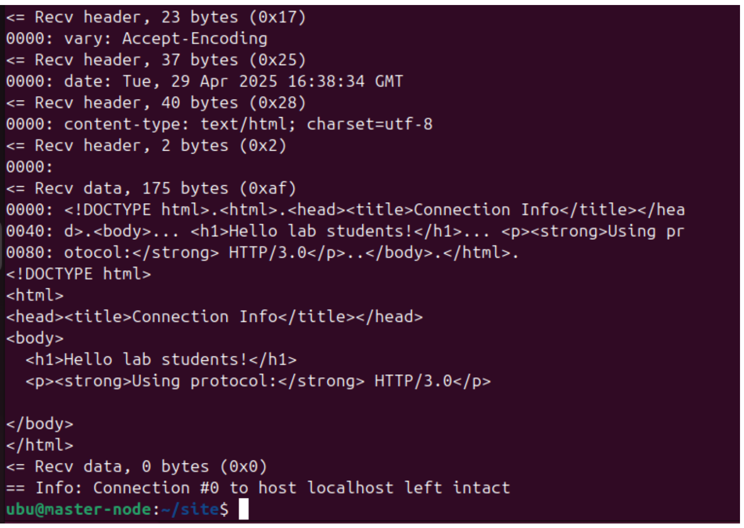


```
  340 397.602689389 127.0.0.1      127.0.0.1      TLSv1.3   190 Application Data
  341 397.602732030 127.0.0.1      127.0.0.1      TLSv1.3   257 Application Data
  342 397.602828667 127.0.0.1      127.0.0.1      TCP        66 52630 → 443 [ACK] Seq=
  343 397.602987471 127.0.0.1      127.0.0.1      TLSv1.3    90 Application Data
  344 397.603051033 127.0.0.1      127.0.0.1      TCP        66 52630 → 443 [FIN, ACK]
  345 397.603282664 127.0.0.1      127.0.0.1      TLSv1.3    90 Application Data
└ 346 397.603297309 127.0.0.1      127.0.0.1      TCP        54 52630 → 443 [RST] Seq=

▸ Frame 341: 257 bytes on wire (2056 bits), 257 bytes captured (2056 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 443, Dst Port: 52630, Seq: 1677, Ack: 800, Len: 191
▸ Transport Layer Security

0050  d8 b4 69 83 7c 6d 80 40  15 77 92 37 89 ff 18 36   ··i·|m·@ ·w·7···6
0060  a2 8a 85 0e 48 02 97 c4  95 75 42 16 9d 7a dc 27   ····H··· ·uB··z·'
0070  3a b0 d3 f3 eb 10 80 ce  c4 eb be c5 53 3e 71 aa   :······· ····S>q·
0080  93 10 75 af f5 02 57 5f  d2 b0 ad 4c f6 e2 8e 48   ··u···W  ···L···H
0090  75 ad ae 42 f2 58 1b 7b  44 55 1b a2 6d 16 89 3c   u··B·X·{ DU··m··<
00a0  2b 31 3e 9d e9 49 f3 62  c1 e9 31 42 1b cb 05 2d   +1>··I·b ··1B···-
00b0  52 ec 34 79 c9 ff 2d e6  73 c2 c2 27 9e f4 85 31   R·4y···· s··'···1
00c0  01 37 33 dc 1c 8a 2c b9  7c 0f c6 b2 04 7f 16 c9   ·73···,· |·······
00d0  ec df da cf 44 e3 3b 11  07 9f bf f0 86 b0 44 b9   ····D·;· ······D·
00e0  58 a7 59 e9 05 37 04 08  df bf f0 c5 10 09 fa 72   X·Y··7·· ·······r

Transport Layer Security (tls), 191 byte(s)      Packets: 346 · Displayed: 346 (100.0%)      Profile: Default
```

Looking at this Wireshark capture of HTTPS traffic using TLS, we can immediately see the huge difference compared to the previous HTTP/1.1 example.

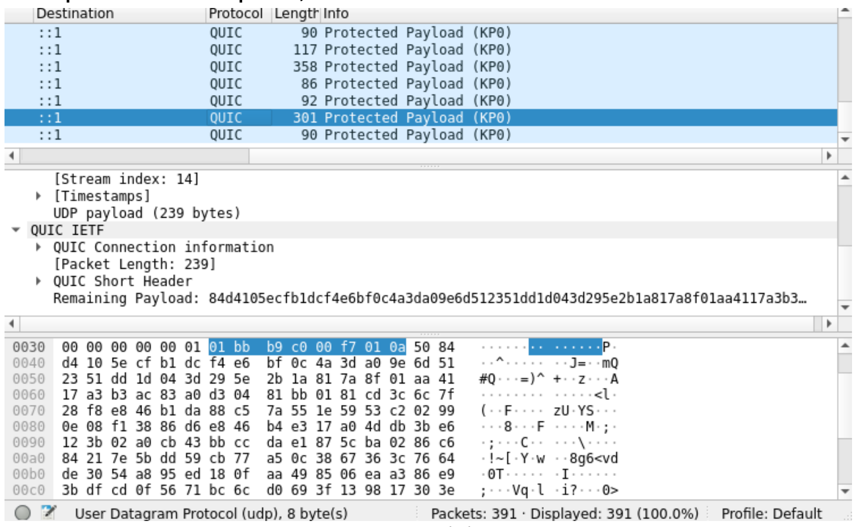
This packet clearly shows encrypted data being transmitted - in the bottom section, all we can see is scrambled, unreadable bytes. The protocol column shows "TLSv1.3" instead of "HTTP," and the packet contents are labeled as "Application Data" rather than showing the actual HTTP content.

6.



```
<= Recv header, 23 bytes (0x17)
0000: vary: Accept-Encoding
<= Recv header, 37 bytes (0x25)
0000: date: Tue, 29 Apr 2025 16:38:34 GMT
<= Recv header, 40 bytes (0x28)
0000: content-type: text/html; charset=utf-8
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 175 bytes (0xaf)
0000: <!DOCTYPE html>.<html>.<head><title>Connection Info</title></hea
0040: d>.<body>... <h1>Hello lab students!</h1>... <p><strong>Using pr
0080: otocol:</strong> HTTP/3.0</p>..</body>.</html>.
<!DOCTYPE html>
<html>
<head><title>Connection Info</title></head>
<body>
  <h1>Hello lab students!</h1>
  <p><strong>Using protocol:</strong> HTTP/3.0</p>

</body>
</html>
<= Recv data, 0 bytes (0x0)
== Info: Connection #0 to host localhost left intact
ubu@master-node:~/site$ 
```

In the provided outputs, we observe clear



```
Destination          Protocol Length Info
 ::1                 QUIC       90 Protected Payload (KP0)
 ::1                 QUIC      117 Protected Payload (KP0)
 ::1                 QUIC      358 Protected Payload (KP0)
 ::1                 QUIC       86 Protected Payload (KP0)
 ::1                 QUIC       92 Protected Payload (KP0)
 ::1                 QUIC      301 Protected Payload (KP0)
 ::1                 QUIC       90 Protected Payload (KP0)

    [Stream index: 14]
  ▸ [Timestamps]
    UDP payload (239 bytes)
▾ QUIC IETF
  ▸ QUIC Connection information
    [Packet Length: 239]
  ▸ QUIC Short Header
    Remaining Payload: 84d4105ecfb1dcf4e6bf0c4a3da09e6d512351dd1d043d295e2b1a817a8f01aa4117a3b3…

0030  00 00 00 00 00 01 01 bb  b9 c0 00 f7 01 0a 50 84   ········ ······P·
0040  d4 10 5e cf b1 dc f4 e6  bf 0c 4a 3d a0 9e 6d 51   ··^····· ··J=··mQ
0050  23 51 dd 1d 04 3d 29 5e  2b 1a 81 7a 8f 01 aa 41   #Q···=)^ +··z···A
0060  17 a3 b3 ac 83 a0 d3 04  81 bb 01 81 cd 3c 6c 7f   ········ ·····<l·
0070  28 f8 e8 46 b1 da 88 c5  7a 55 1e 59 53 c2 02 99   (··F···· zU·YS···
0080  0e 08 f1 38 86 d6 e8 46  b4 e3 17 a0 4d db 3b e6   ···8···F ····M·;·
0090  12 3b 02 a0 cb 43 bb cc  da e1 87 5c ba 02 86 c6   ·;···C·· ···\····
00a0  84 21 7e 5b dd 59 cb 77  a5 0c 38 67 36 3c 76 64   ·!~[·Y·w ··8g6<vd
00b0  de 30 54 a8 95 ed 18 0f  aa 49 85 06 ea a3 86 e9   ·0T····· ·I······
00c0  3b df cd 0f 56 71 bc 6c  d0 69 3f 13 98 17 30 3e   ;···Vq·l ·i?···0>

User Datagram Protocol (udp), 8 byte(s)      Packets: 391 · Displayed: 391 (100.0%)      Profile: Default
```

differences between using the curl command (with the `--trace-ascii` flag) and inspecting HTTP/3 traffic in Wireshark. The curl output directly shows readable details like HTTP headers (such as content-type, content-length, and date) and the actual HTML content returned by the server. On the other hand, the Wireshark output only displays QUIC packets labeled as "Protected Payload," indicating that the data is

encrypted and not directly readable. This means curl gives an easy-to-understand view of the decrypted communication, whereas Wireshark captures the low-level, encrypted traffic, requiring additional steps or keys to interpret clearly.
In addition, curl explicitly mentions HTTP/3 protocol details, whereas Wireshark primarily identifies the underlying QUIC protocol.

7.

i. curl:

--http2: Force the client to use the HTTP/2 protocol when communicating with the server.

-k: Allow insecure SSL connections (skip certificate verification since we are using a local self-signed server).

--parallel --parallel-immediate: Fetch multiple URLs in parallel **immediately** instead of waiting for one to finish before starting the next.

--output slow1_downloaded.txt …: Save each downloaded file to a specific output file name (one output per URL).

--trace-ascii trace_http2.log: Record a detailed ASCII trace of all sent/received HTTP/2 frames into trace_http2.log.

--trace-time: Add timestamps to every line inside the trace file, so the timing of events can be analyzed.

https://localhost/slow1.dat https://localhost/slow2.dat https://localhost/fast.html: These are the three files being requested from the server over HTTPS on localhost.

This command downloads all three files at once over HTTP/2, records the entire session into a log file, and saves the files separately.

grep:

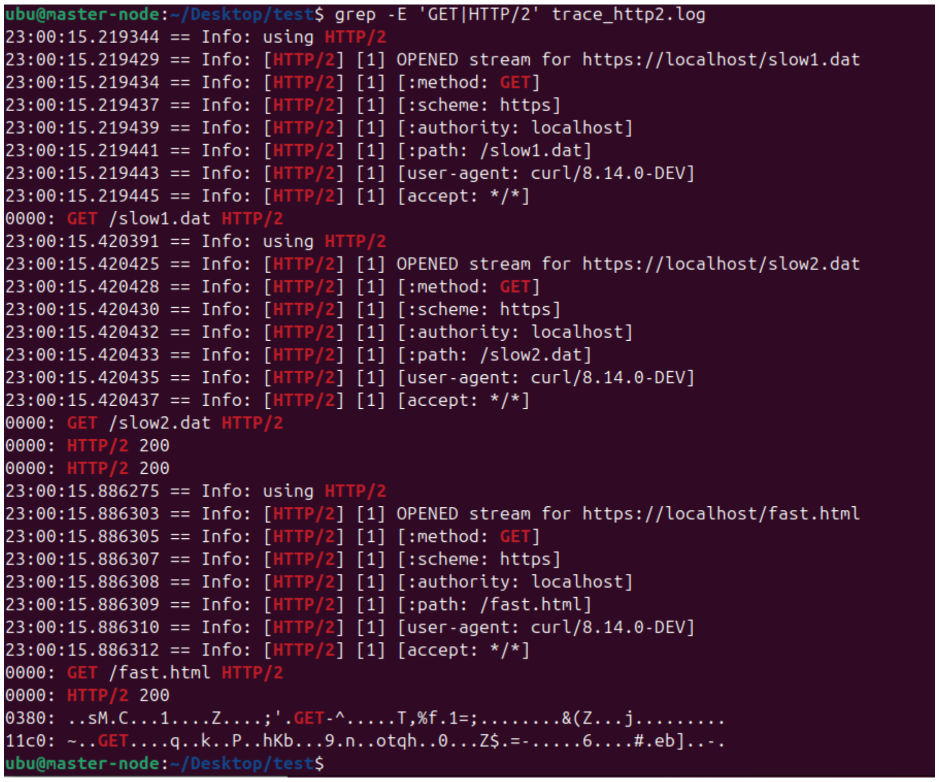
Searches inside the trace file for lines that contain either:

- the HTTP GET requests (showing what was requested),
- or the HTTP/2 responses (showing when a response was received).

-E: Use extended regular expressions so we can match multiple patterns (GET or HTTP/2).

This allows filtering only the important parts (requests and responses) from the trace.

ii.



```
ubu@master-node:~/Desktop/test$ grep -E 'GET|HTTP/2' trace_http2.log
23:00:15.219344 == Info: using HTTP/2
23:00:15.219429 == Info: [HTTP/2] [1] OPENED stream for https://localhost/slow1.dat
23:00:15.219434 == Info: [HTTP/2] [1] [:method: GET]
23:00:15.219437 == Info: [HTTP/2] [1] [:scheme: https]
23:00:15.219439 == Info: [HTTP/2] [1] [:authority: localhost]
23:00:15.219441 == Info: [HTTP/2] [1] [:path: /slow1.dat]
23:00:15.219443 == Info: [HTTP/2] [1] [user-agent: curl/8.14.0-DEV]
23:00:15.219445 == Info: [HTTP/2] [1] [accept: */*]
0000: GET /slow1.dat HTTP/2
23:00:15.420391 == Info: using HTTP/2
23:00:15.420425 == Info: [HTTP/2] [1] OPENED stream for https://localhost/slow2.dat
23:00:15.420428 == Info: [HTTP/2] [1] [:method: GET]
23:00:15.420430 == Info: [HTTP/2] [1] [:scheme: https]
23:00:15.420432 == Info: [HTTP/2] [1] [:authority: localhost]
23:00:15.420433 == Info: [HTTP/2] [1] [:path: /slow2.dat]
23:00:15.420435 == Info: [HTTP/2] [1] [user-agent: curl/8.14.0-DEV]
23:00:15.420437 == Info: [HTTP/2] [1] [accept: */*]
0000: GET /slow2.dat HTTP/2
0000: HTTP/2 200
0000: HTTP/2 200
23:00:15.886275 == Info: using HTTP/2
23:00:15.886303 == Info: [HTTP/2] [1] OPENED stream for https://localhost/fast.html
23:00:15.886305 == Info: [HTTP/2] [1] [:method: GET]
23:00:15.886307 == Info: [HTTP/2] [1] [:scheme: https]
23:00:15.886308 == Info: [HTTP/2] [1] [:authority: localhost]
23:00:15.886309 == Info: [HTTP/2] [1] [:path: /fast.html]
23:00:15.886310 == Info: [HTTP/2] [1] [user-agent: curl/8.14.0-DEV]
23:00:15.886312 == Info: [HTTP/2] [1] [accept: */*]
0000: GET /fast.html HTTP/2
0000: HTTP/2 200
0380: ..sM.C...1....Z....;'.GET-^.....T,%f.1=;........&(Z...j.........
11c0: ~..GET....q..k..P..hKb...9.n..otqh..0...Z$.=-.....6....#.eb]..-.
ubu@master-node:~/Desktop/test$
```

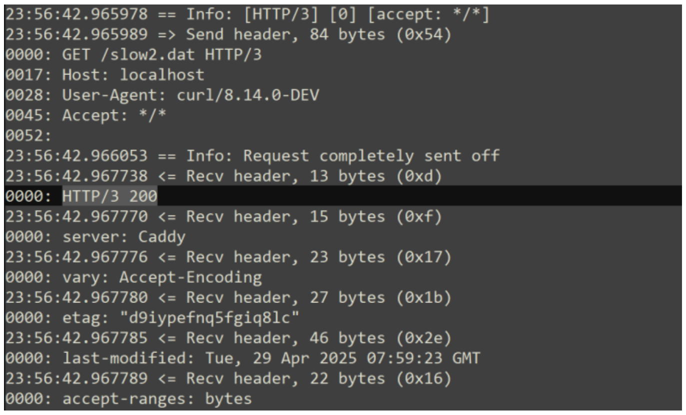
Although HTTP/2 supports multiplexing multiple streams over a single TCP connection, it still relies on TCP for transport. TCP ensures strict in-order delivery of packets — meaning if one packet is lost or delayed, all following packets must wait until the missing packet is received.

In our experiment, the two large files (slow1.dat and slow2.dat) were requested first. Since they are big, they occupy a lot of TCP packets. Even though fast.html is very small and requested afterward, its packets are queued behind the packets of the large files at the TCP layer.

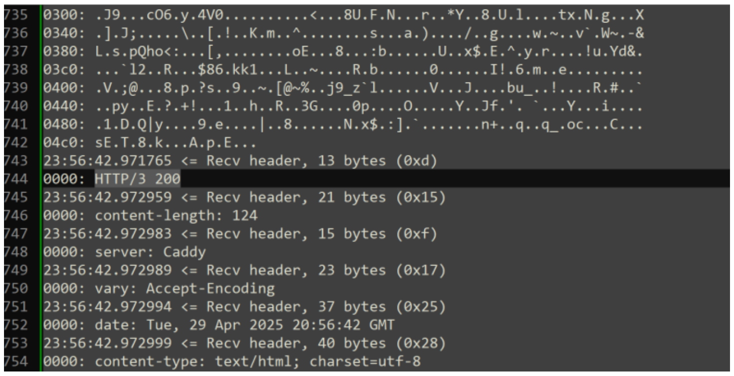
If even one packet from slow1.dat or slow2.dat is delayed or lost, TCP holds back all subsequent packets (including the ones for fast.html) until the missing packet is retransmitted and received. This causes fast.html to be delayed even though it could have been transmitted earlier if TCP allowed out-of-order delivery.

Thus, TCP-level HOL blocking prevents HTTP/2 from delivering small, fast responses quickly, despite HTTP/2's multiplexing features.

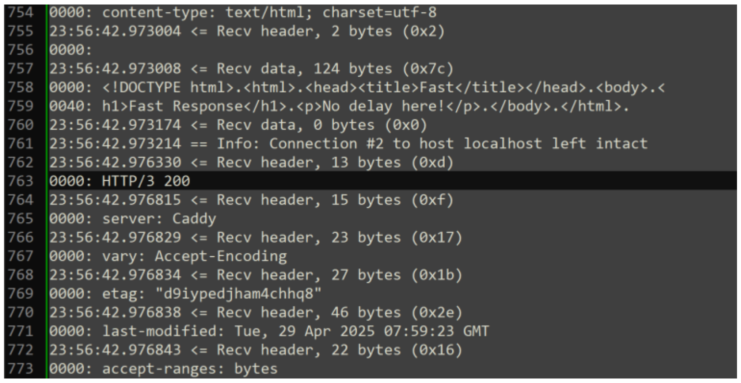
8.



```
23:56:42.965978 == Info: [HTTP/3] [0] [accept: */*]
23:56:42.965989 => Send header, 84 bytes (0x54)
0000: GET /slow2.dat HTTP/3
0017: Host: localhost
0028: User-Agent: curl/8.14.0-DEV
0045: Accept: */*
0052:
23:56:42.966053 == Info: Request completely sent off
23:56:42.967738 <= Recv header, 13 bytes (0xd)
0000: HTTP/3 200
23:56:42.967770 <= Recv header, 15 bytes (0xf)
0000: server: Caddy
23:56:42.967776 <= Recv header, 23 bytes (0x17)
0000: vary: Accept-Encoding
23:56:42.967780 <= Recv header, 27 bytes (0x1b)
0000: etag: "d9iypefnq5fgiq8lc"
23:56:42.967785 <= Recv header, 46 bytes (0x2e)
0000: last-modified: Tue, 29 Apr 2025 07:59:23 GMT
23:56:42.967789 <= Recv header, 22 bytes (0x16)
0000: accept-ranges: bytes
```



```
0300: .J9...cO6.y.4V0..........<...8U.F.N...r..*Y..8.U.l....tx.N.g...X
0340: .].J;.....\..[.!..K.m..^........s...a.)..../..g....w.~..v`.W~.-&
0380: L.s.pQho<:...[,.........oE...8...:b......U..x$.E.^.y.r....!u.Yd&.
03c0: ...`12..R...$86.kk1...L..~....R.b......0......I!.6.m..e.........
0400: .V.;@...8.p.?s..9..~.[@~%..j9_z`l......V...J....bu_..!....R.#..`
0440: ..py..E.?.+!...1..h..R..3G....0p....O.....Y..Jf.'. `...Y...i....
0480: .1.D.Q|y....9.e....|..8......N.x$.:].`.......n+..q..q_.oc...C...
04c0: sE.T.8.k...A.p.E...
23:56:42.971765 <= Recv header, 13 bytes (0xd)
0000: HTTP/3 200
23:56:42.972959 <= Recv header, 21 bytes (0x15)
0000: content-length: 124
23:56:42.972983 <= Recv header, 15 bytes (0xf)
0000: server: Caddy
23:56:42.972989 <= Recv header, 23 bytes (0x17)
0000: vary: Accept-Encoding
23:56:42.972994 <= Recv header, 37 bytes (0x25)
0000: date: Tue, 29 Apr 2025 20:56:42 GMT
23:56:42.972999 <= Recv header, 40 bytes (0x28)
0000: content-type: text/html; charset=utf-8
```

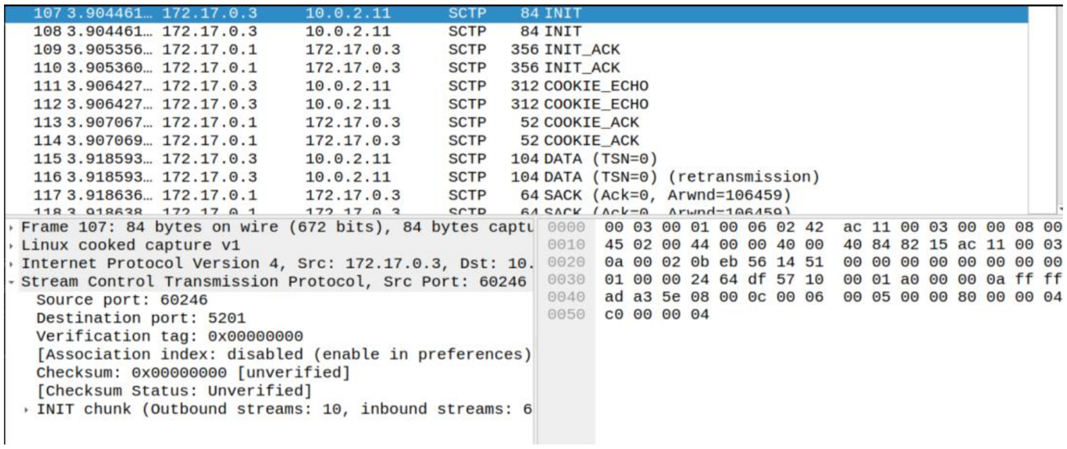


```
0000: content-type: text/html; charset=utf-8
23:56:42.973004 <= Recv header, 2 bytes (0x2)
0000:
23:56:42.973008 <= Recv data, 124 bytes (0x7c)
0000: <!DOCTYPE html>.<html>.<head><title>Fast</title></head>.<body>.<
0040: h1>Fast Response</h1>.<p>No delay here!</p>.</body>.</html>.
23:56:42.973174 <= Recv data, 0 bytes (0x0)
23:56:42.973214 == Info: Connection #2 to host localhost left intact
23:56:42.976330 <= Recv header, 13 bytes (0xd)
0000: HTTP/3 200
23:56:42.976815 <= Recv header, 15 bytes (0xf)
0000: server: Caddy
23:56:42.976829 <= Recv header, 23 bytes (0x17)
0000: vary: Accept-Encoding
23:56:42.976834 <= Recv header, 27 bytes (0x1b)
0000: etag: "d9iypedjham4chhq8"
23:56:42.976838 <= Recv header, 46 bytes (0x2e)
0000: last-modified: Tue, 29 Apr 2025 07:59:23 GMT
23:56:42.976843 <= Recv header, 22 bytes (0x16)
0000: accept-ranges: bytes
```

In the provided trace logs, all three resources (slow1.dat, slow2.dat, fast.html) were requested in close succession and responded to with similar timestamps, regardless of their order. This demonstrates that HTTP/3 eliminates TCP-level Head-of-Line blocking, enabling the fast.html response to be delivered without delay, even while larger files (slow1/slow2) are in progress.

These nearly simultaneous timestamps indicate that all three streams were handled independently, and the small file fast.html was not delayed by the slower file downloads. This is because HTTP/3 runs over QUIC and UDP, which supports independent multiplexing per stream and avoids the TCP HOL blocking seen in HTTP/2.

9.



```
107 3.904461… 172.17.0.3    10.0.2.11     SCTP    84 INIT
108 3.904461… 172.17.0.3    10.0.2.11     SCTP    84 INIT
109 3.905356… 172.17.0.1    172.17.0.3    SCTP   356 INIT_ACK
110 3.905360… 172.17.0.1    172.17.0.3    SCTP   356 INIT_ACK
111 3.906427… 172.17.0.3    10.0.2.11     SCTP   312 COOKIE_ECHO
112 3.906427… 172.17.0.3    10.0.2.11     SCTP   312 COOKIE_ECHO
113 3.907067… 172.17.0.1    172.17.0.3    SCTP    52 COOKIE_ACK
114 3.907069… 172.17.0.1    172.17.0.3    SCTP    52 COOKIE_ACK
115 3.918593… 172.17.0.3    10.0.2.11     SCTP   104 DATA (TSN=0)
116 3.918593… 172.17.0.3    10.0.2.11     SCTP   104 DATA (TSN=0) (retransmission)
117 3.918636… 172.17.0.1    172.17.0.3    SCTP    64 SACK (Ack=0, Arwnd=106459)
118 3.918638  172.17.0.1    172.17.0.3    SCTP    64 SACK (Ack=0, Arwnd=106459)
```

```
› Frame 107: 84 bytes on wire (672 bits), 84 bytes captu     0000  00 03 00 01 00 06 02 42  ac 11 00 03 00 00 08 00
› Linux cooked capture v1                                    0010  45 02 00 44 00 00 40 00  40 84 82 15 ac 11 00 03
› Internet Protocol Version 4, Src: 172.17.0.3, Dst: 10.     0020  0a 00 02 0b eb 56 14 51  00 00 00 00 00 00 00 00
⌄ Stream Control Transmission Protocol, Src Port: 60246      0030  01 00 00 24 64 df 57 10  00 01 a0 00 00 0a ff ff
    Source port: 60246                                       0040  ad a3 5e 08 00 0c 00 06  00 05 00 00 80 00 00 04
    Destination port: 5201                                   0050  c0 00 00 04
    Verification tag: 0x00000000
    [Association index: disabled (enable in preferences)
    Checksum: 0x00000000 [unverified]
    [Checksum Status: Unverified]
  › INIT chunk (Outbound streams: 10, inbound streams: 6
```

The SCTP handshake, as observed above, consists of four distinct steps:

INIT – The client begins the association by sending an INIT chunk.
INIT_ACK – The server responds with an INIT_ACK.
COOKIE_ECHO – The client then replies with a COOKIE_ECHO, echoing a cookie sent by the server in the INIT_ACK.
COOKIE_ACK – The server finalizes the handshake by sending a COOKIE_ACK.

This 4-way handshake is designed with a built-in cookie mechanism for protection against SYN flood attacks, unlike TCP. The client must return the server's cookie to prove its identity before the server allocates resources, making SCTP more resilient. By contrast, TCP (as used in HTTP/2) uses a 3-way handshake:

SYN – Client initiates the connection.
SYN-ACK – Server acknowledges and responds.
ACK – Client acknowledges back.

TCP's handshake is shorter but does not include a cookie verification mechanism, making it more vulnerable to spoofed connections and certain DoS attacks.

10. We don't see HEARTBEAT packets in the capture because they're only used when there's no data being sent for a while or when SCTP uses multiple paths (multi-homing). In our case, the connection was active with constant data transfer and only one path was used, so there was no need for HEARTBEAT messages.

11. QUIC, used in HTTP/3, runs over UDP, while SCTP is its own transport protocol built on IP. Both support sending multiple streams within a single connection, but SCTP

calls this "multistreaming" and QUIC calls it "multiplexing". QUIC includes built-in encryption and better NAT traversal, while SCTP is more focused on reliability and message boundaries. In terms of congestion control, both implement their own mechanisms, but QUIC is more optimized for modern web performance.