COL 334/672 Computer Networks
Assignment 1: Getting To Know Network Traffic

Saharsh Laud
Enrollment Number: 2024MCS2002

Deadline: August 22, 2025

Contents

1 Measurement Tools

1.1 Ping

Problem Statement:
Ping the following two websites: `google.com` and `craigslist.com`. You should ping these websites 10 times and attach screenshots for each case.

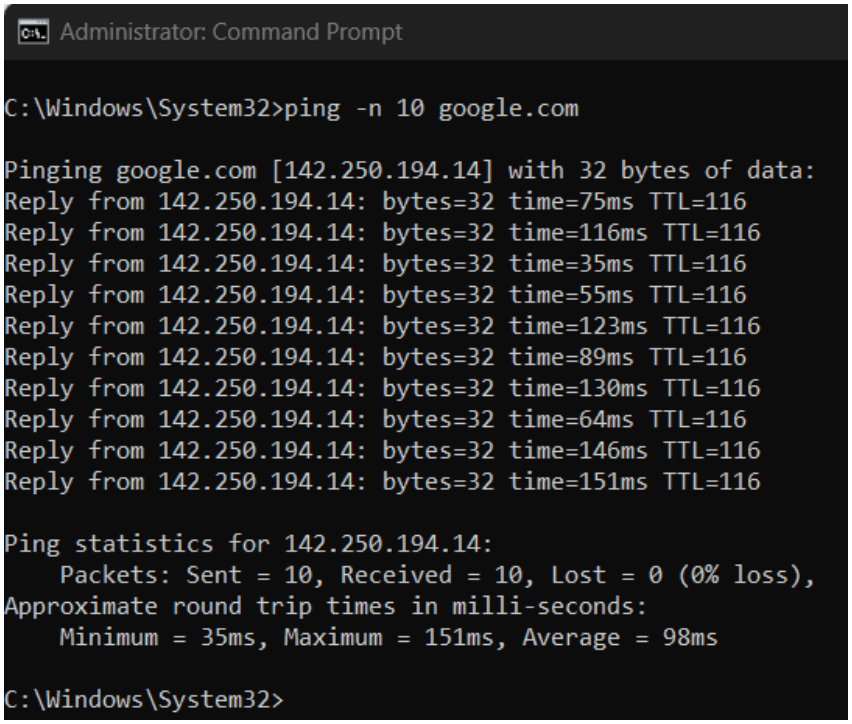


Figure 1: Pinging google.com

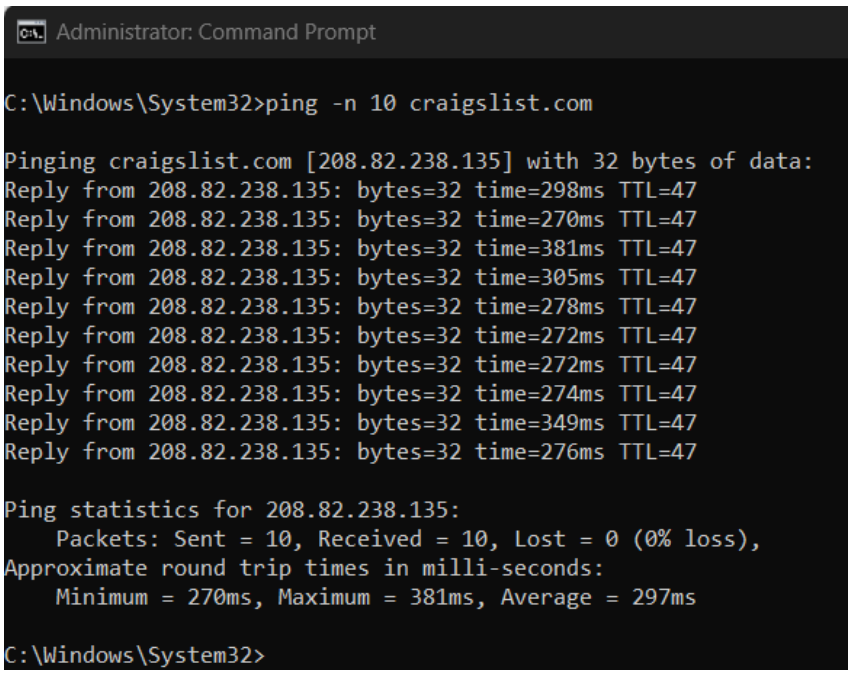


Figure 2: Pinging craigslist.com

A. **Explain the protocol(s) being used by the ping tool. Where does it sit in the network protocol stack?**

Answer:
The `ping` tool uses the Internet Control Message Protocol (ICMP) to check reachability and measure round-trip time (RTT). It sends an *ICMP Echo Request* and receives an *ICMP Echo Reply* if the host is reachable (ICMPv4 for IPv4, ICMPv6 for IPv6).
In the 5-layer Internet stack: the `ping` program runs at the **Application** layer; it does *not* use TCP/UDP at the Transport layer; ICMP operates at the **Network layer** inside IP; Link and Physical layers carry the frames/bits over each hop.

B. **Compare the average ping latencies for the two websites from the same network. Why might they differ? Why might latency across multiple pings for the same website differ?**

Answer:
On the same Wi-Fi network, we measured:
`google.com` (IPv4): average RTT = 98 ms (min 35 ms, max 151 ms).
`craigslist.com` (IPv4): average RTT = 297 ms (min 270 ms, max 381 ms).

Why the averages differ:

- **Server location/distance:** Google serves from a nearby India edge, so packets travel a shorter distance. Craigslist is hosted in the US, so packets take a longer route, adding delay.
- **Path taken through the network:** Different routes and more devices along the way add small delays that add up. Fewer/shorter paths tend to have lower RTT.

Why the same site's RTT varies across pings:

- **Real-time traffic (congestion):** Internet traffic changes constantly, which changes queueing delay.
- **Wi-Fi conditions:** Local wireless interference/retries can add small, variable delays.
- **Server responsiveness:** The destination or intermediate devices may respond slightly slower at times.

In short, `google.com` is closer on the network and responds faster; `craigslist.com` is farther and takes longer.

C. **Force ping to use IPv6 for both websites. Explain how you did it and whether it worked. If IPv6 fails, explain why?**

Answer:
We forced IPv6 using `ping -6 <hostname>` and checked DNS using `nslookup <hostname>`.
- *google.com (IPv6):* **Worked**

We received replies from an IPv6 address (2404:6800:4002:81e::200e) with an average RTT of about 31 ms. This shows that the force ping to IPv6 is working

end-to-end.

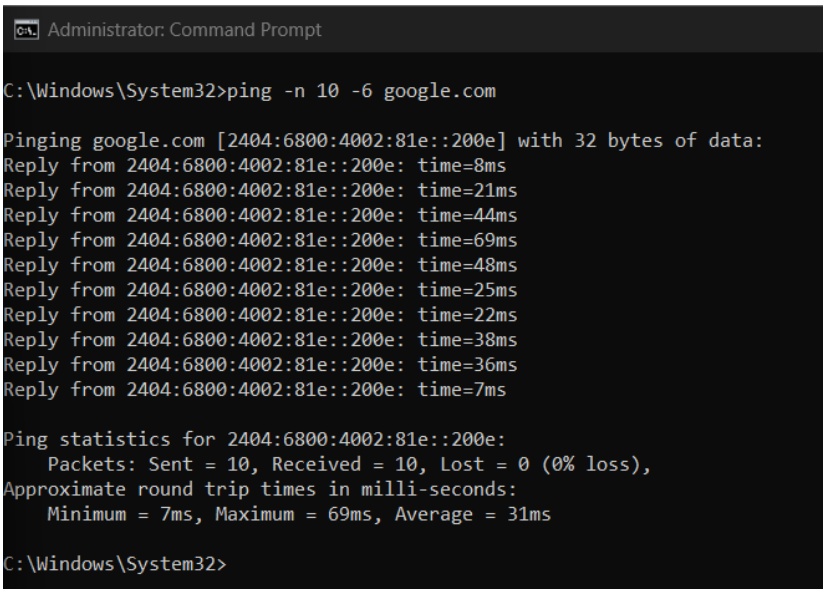


Figure 3: Force Pinging google.com over IPv6

- *craigslist.com (IPv6):* **Did not work.**

In our DNS lookups, `nslookup` did not return an IPv6 address for `craigslist.com`, so `ping -6` had no IPv6 target to reach. Since IPv6 to `google.com` works on the same host and network, this indicates that `craigslist.com` was not providing an IPv6 address in our queries at the time of testing (i.e., no IPv6 configuration visible), rather than an issue with our local IPv6 setup.

Figure 4: Force Pinging craigslist.com over IPv6

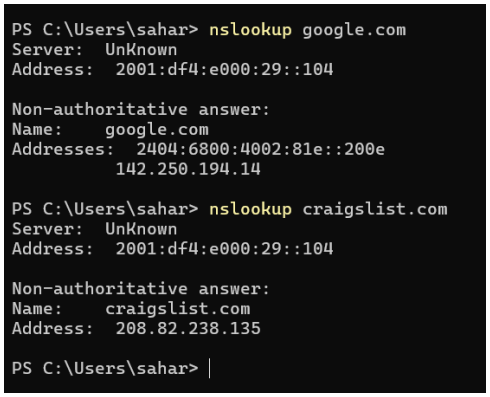


Figure 5: nslookup for google.com returned IPv4 and IPv6 addresses but craigslist.com did not return an IPv6 address

D. **Ping allows you to specify the size of packets to send. What is the maximum size of ping packets that you are able to send? Explain.**

Answer:
Using the `-l` option, we can set the ping data size up to maximum of **65,500 bytes**. Our outputs also show the tool accepts values in the range 0–65,500.

In practice, very large pings did not get replies on our path:

- `-l 65000` and `-l 65500` timed out (no reply).

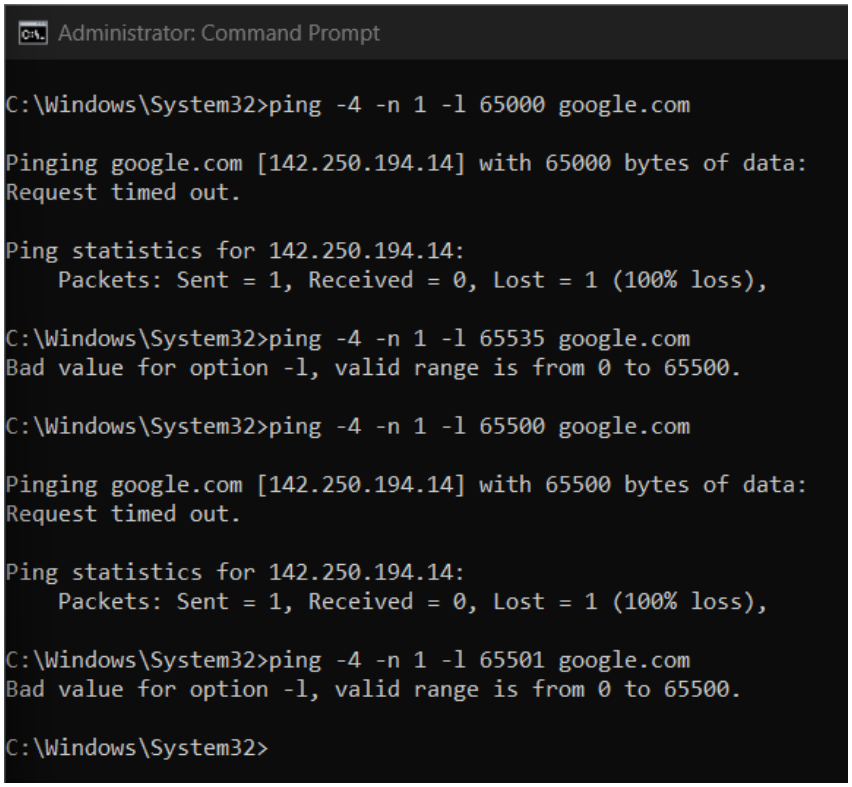


Figure 6: Large ping sizes (65,000/65,500 B) timed out on our path.

This is because the network path has a **Maximum Transmission Unit (MTU)**. If a packet is larger than the MTU, devices may fragment it or drop it, so it may never reach the destination or return a reply. To check the path limit, we tested sizes near the MTU:

- **IPv4 (no fragmentation allowed):** `-f -l 1472` succeeded; `-f -l 1473` failed. This shows that 1,500 B MTU is in use on the path, and the largest ICMP payload that works without fragmentation is **1472 bytes**.
- **IPv6:** `-l 1452` succeeded; `-l 1453` timed out. This matches a 1,500 B MTU, with the largest ICMPv6 payload that works being **1452 bytes**.

So:

- **Maximum size you can set in ping:** 65,500 B.
- **Largest sizes that actually worked on our path:** 1,472 B (IPv4, no fragmentation) and 1,452 B (IPv6).

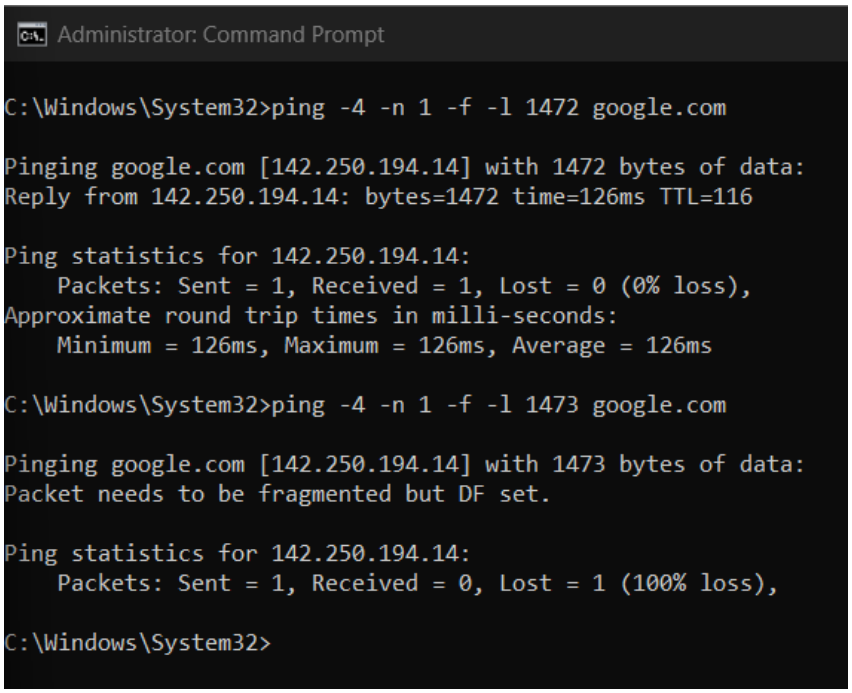


Figure 7: IPv4 MTU check: 1,472 B payload succeeded with no fragmentation; 1,473 B failed.

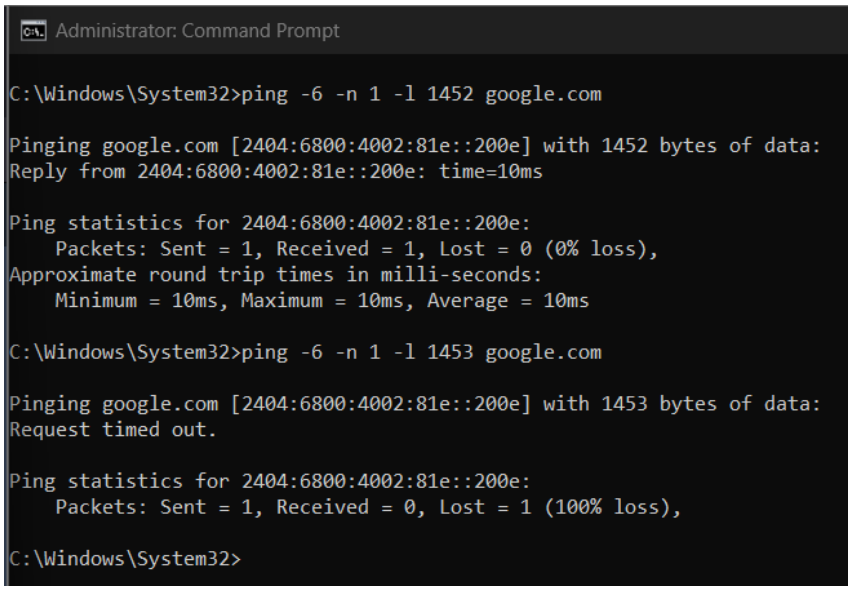


Figure 8: IPv6 MTU check: 1,452 B payload succeeded; 1,453 B timed out.

1.2 Traceroute

Problem Statement:
Once you've seen how long it takes to get there, let's find out how you get there. Log the server IP addresses for the two websites in above case. Use traceroute to find the path taken by the packets in both cases and attach the screenshot.

A. Mention the number of IP hops in each case. List the autonomous systems you pass through (Hint: Online tools can map IPs to AS numbers.).

Answer:
Here are the screenshots for the traceroute and the AS number lookups:

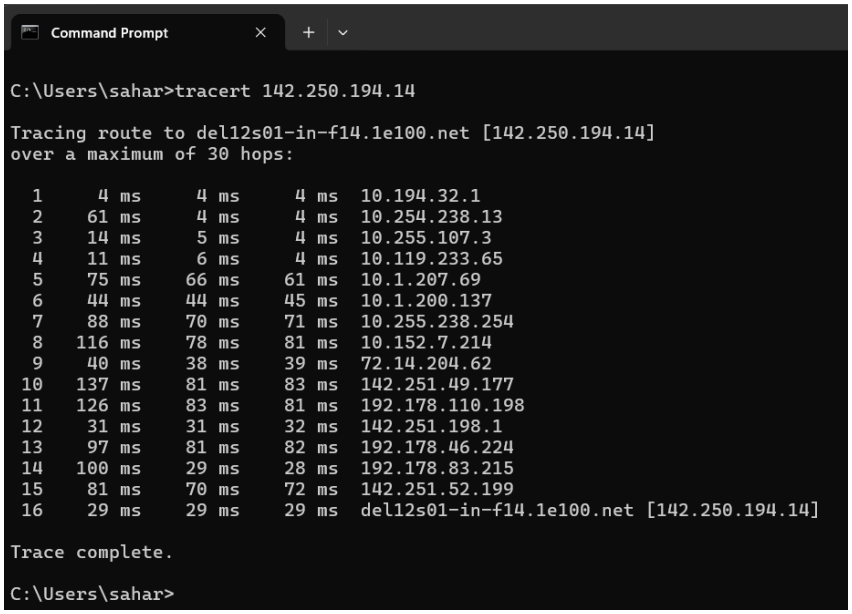


Figure 9: Traceroute to google.com (IPv4) with server IP: 142.250.194.14.

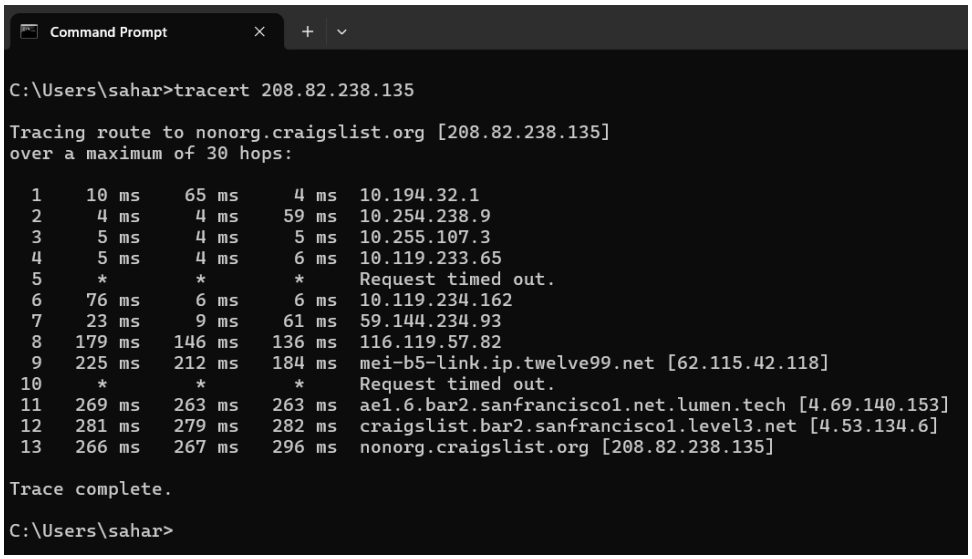
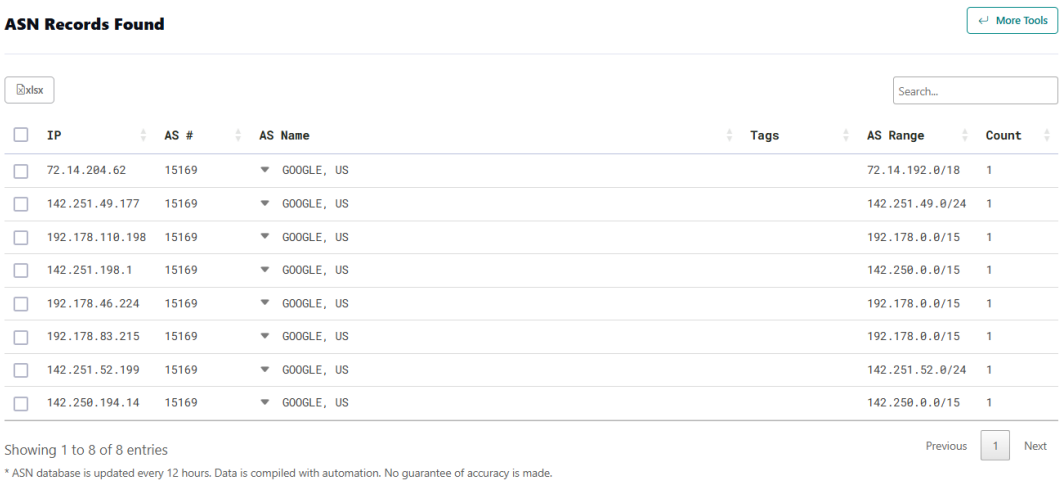


Figure 10: Traceroute to craigslist.com (IPv4) with server IP: 208.82.238.135.



ASN Records Found

More Tools

xlsx

Search...

IP	AS #	AS Name	Tags	AS Range	Count
72.14.204.62	15169	GOOGLE, US		72.14.192.0/18	1
142.251.49.177	15169	GOOGLE, US		142.251.49.0/24	1
192.178.110.198	15169	GOOGLE, US		192.178.0.0/15	1
142.251.198.1	15169	GOOGLE, US		142.250.0.0/15	1
192.178.46.224	15169	GOOGLE, US		192.178.0.0/15	1
192.178.83.215	15169	GOOGLE, US		192.178.0.0/15	1
142.251.52.199	15169	GOOGLE, US		142.251.52.0/24	1
142.250.194.14	15169	GOOGLE, US		142.250.0.0/15	1

Showing 1 to 8 of 8 entries

Previous 1 Next

* ASN database is updated every 12 hours. Data is compiled with automation. No guarantee of accuracy is made.

Figure 11: ASN lookup for public hops towards google.com. All public hops shown are in AS15169 (GOOGLE).

ASN Records Found

More Tools

xlsx

Search...

IP	AS #	AS Name	Tags	AS Range	Count
59.144.234.93	9498	BBIL-AP BHARTI Airtel Ltd., IN		59.144.234.0/24	1
116.119.57.82	9498	BBIL-AP BHARTI Airtel Ltd., IN		116.119.32.0/19	1
62.115.42.118	1299	TWELVE99 Arelion, fka Telia Carrier, SE		62.115.0.0/16	1
4.69.140.153	3356	LEVEL3, US		4.0.0.0/9	1
4.53.134.6	3356	LEVEL3, US		4.0.0.0/9	1
208.82.238.135	22414	CRAIGS-NET-1, US		208.82.238.0/24	1

Showing 1 to 6 of 6 entries

Previous 1 Next

* ASN database is updated every 12 hours. Data is compiled with automation. No guarantee of accuracy is made.

Figure 12: ASN lookup for public hops towards craigslist.com: AS9498 (Bharti Airtel) → AS1299 (Arelion/Twelve99) → AS3356 (Lumen/Level3) → AS22414 (CRAIGS-NET-1).

- *google.com:* **Number of Hops = 16**
From our traceroute, the destination was reached at hop 16 (142.250.194.14). Private 10.x.x.x hops belong to the campus internal network. All public hops observed after that are owned by Google.
Autonomous systems passed through: **AS15169 (GOOGLE)**.

- *craigslist.com:* **Number of Hops = 13**
From our traceroute, the destination was reached at hop 13 (208.82.238.135). One hop showed no reply (), but later hops responded.
Autonomous systems passed through: **AS9498 (Bharti Airtel) → AS1299 (Arelion/Twelve99) → AS3356 (Lumen/Level3) → AS22414 (CRAIGS-NET-1)**.
(Private 10.x.x.x hops are internal and do not have public ASNs.)

B. Did you see "*" in the traceroute output? Explain what it means.

Answer:

Yes, in the traceroute for `craigslist.com` we saw "*" at some hop(s). A "*" means that hop did not send a reply within the timeout. In traceroute, probes are sent with increasing TTL (Time To Live), and each router is expected to reply with an ICMP "Time Exceeded." If a router blocks or ignores these diagnostic replies, rate-limits them, or the network is briefly congested, no reply comes back in time, so "*" is shown. This does not mean the path is broken as later hops and the destination still responded.

C. Did you observe multiple IP addresses for the same hop count? If yes, explain the reason.

Answer:
Yes. While running `traceroute` multiple times, we sometimes see different IP addresses for the same hop. This happens because routers use dynamic routing and there can be more than one valid path to the destination. If there is congestion on the usual path, packets may be sent via an alternate path. In general, different probes can take slightly different routes depending on the network state at that moment.

```
sash_unix@Saharsh:~$ traceroute google.com
traceroute to google.com (142.250.194.14), 30 hops max, 60 byte packets
 1  Saharsh.mshome.net (172.20.0.1)  0.661 ms  0.524 ms  0.601 ms
 2  10.194.32.1 (10.194.32.1)  5.816 ms  5.800 ms  5.786 ms
 3  10.254.238.5 (10.254.238.5)  5.860 ms  5.847 ms 10.254.238.13 (10.254.238.13)  5.902 ms
 4  10.255.107.3 (10.255.107.3)  5.814 ms  5.794 ms  5.777 ms
 5  10.119.233.65 (10.119.233.65)  5.755 ms  5.738 ms  5.721 ms
 6  10.1.207.69 (10.1.207.69)  38.428 ms  38.007 ms  40.309 ms
 7  10.1.200.137 (10.1.200.137)  55.906 ms  53.428 ms  53.403 ms
 8  10.255.237.94 (10.255.237.94)  55.190 ms 10.255.238.122 (10.255.238.122)  39.271 ms  39.232 ms
 9  10.152.7.214 (10.152.7.214)  44.743 ms  39.187 ms  55.042 ms
10  * * *
11  * * *
12  142.251.64.10 (142.251.64.10)  39.621 ms 142.251.64.8 (142.251.64.8)  55.298 ms  38.702 ms
13  142.250.226.134 (142.250.226.134)  55.211 ms 192.178.110.206 (192.178.110.206)  48.492 ms  49.585 ms
14  * 142.251.198.1 (142.251.198.1)  35.678 ms *
15  192.178.252.124 (192.178.252.124)  30.797 ms 192.178.252.112 (192.178.252.112)  29.132 ms 192.178.252.116 (192.178.2
52.116)  33.649 ms
16  216.239.62.181 (216.239.62.181)  30.727 ms 216.239.54.93 (216.239.54.93)  33.607 ms 192.178.83.225 (192.178.83.225)
 46.389 ms
17  142.251.52.199 (142.251.52.199)  40.078 ms  46.316 ms  76.727 ms
18  del12s01-in-f14.1e100.net (142.250.194.14)  75.194 ms  75.172 ms  72.238 ms
```

Figure 13: Traceroute example showing multiple IPs at the same hop number.

D. Try to geolocate the IP addresses. You can use two different methods: First, try doing the reverse DNS lookup on the IP address and see if you can infer the location from the DNS address. If the reverse DNS lookup fails, use the Maxmind database for IP geolocation. Note the IP geolocation can sometimes be wrong, especially if you are using the Maxmind database. In fact, accurate geolocation of IP addresses is still an active area of research. Now compare the geographical path with the observed RTTs. Do these intuitively make sense? Explain why.

Answer:
We used two methods to find the geographic locations of IP addresses along our traceroute paths:

Method 1: Reverse DNS Lookup
We used `nslookup <ip>` to check if the IP addresses had reverse DNS records (also called PTR records). These records sometimes contain location hints like city codes or datacenter names in the hostname.

For **Google path hops**, most intermediate IPs did not return useful hostnames (showing "Non-existent domain"). However, the destination IP 142.250.194.14 resolved to `del12s01-in-f14.1e100.net`, where "del" likely indicates Delhi, India.

```
C:\Users\sahar>nslookup 72.14.204.62
Server:  UnKnown
Address:  2001:df4:e000:29::104

*** UnKnown can't find 72.14.204.62: Non-existent domain

C:\Users\sahar>nslookup 142.251.49.177
Server:  UnKnown
Address:  2001:df4:e000:29::104

*** UnKnown can't find 142.251.49.177: Non-existent domain

C:\Users\sahar>nslookup 192.178.110.198
Server:  UnKnown
Address:  2001:df4:e000:29::104

*** UnKnown can't find 192.178.110.198: Non-existent domain

C:\Users\sahar>nslookup 142.250.194.14
Server:  UnKnown
Address:  2001:df4:e000:29::104

Name:    del12s01-in-f14.1e100.net
Address:  142.250.194.14
```

Figure 14: Reverse DNS lookup results for Google hops showing most return "Non-existent domain" except destination which resolves to del12s01-in-f14.1e100.net (Delhi indicator).

For **Craigslist path hops**, we found some useful hostnames:

- 62.115.42.118 → `mei-b5-link.ip.twelve99.net` (transit hop)
- 4.69.140.153 → `ae1.6.bar2.sanfrancisco1.net.lumen.tech` (clearly San Francisco)
- 4.53.134.6 → `craigslist.bar2.sanfrancisco1.level3.net` (San Francisco handoff to Craigslist)

```
C:\Users\sahar>nslookup 59.144.234.93
Server:  UnKnown
Address:  2001:df4:e000:29::104

DNS request timed out.
    timeout was 2 seconds.
*** Request to UnKnown timed-out

C:\Users\sahar>nslookup 62.115.42.118
Server:  UnKnown
Address:  2001:df4:e000:29::104

Name:    mei-b5-link.ip.twelve99.net
Address:  62.115.42.118


C:\Users\sahar>nslookup 4.69.140.153
Server:  UnKnown
Address:  2001:df4:e000:29::104

Name:    ae1.6.bar2.sanfrancisco1.net.lumen.tech
Address:  4.69.140.153


C:\Users\sahar>nslookup 4.53.134.6
Server:  UnKnown
Address:  2001:df4:e000:29::104

Name:    craigslist.bar2.sanfrancisco1.level3.net
Address:  4.53.134.6


C:\Users\sahar>nslookup 208.82.238.135
Server:  UnKnown
Address:  2001:df4:e000:29::104

Name:    nonorg.craigslist.org
Address:  208.82.238.135
```

Figure 15: Reverse DNS lookup results for Craigslist path showing useful location hints in hostnames like sanfrancisco1.net.lumen.tech and twelve99.net.

Method 2: MaxMind Database

We used the MaxMind geolocation database, but as noted in the assignment, geolocation can sometimes be wrong and is still an active research area.

IP Address	Location	Network	Postal Code	Approximate Latitude / Longitude*, and Accuracy Radius	ISP / Organization	Domain
72.14.204.62	United States (US), North America	72.14.204.0/22	-	37.751, -97.822 (1000 km)	Google	-
142.251.49.177	United States (US), North America	142.251.48.0/20	-	37.751, -97.822 (1000 km)	Google	-
192.178.110.198	United States (US), North America	192.178.110.0/24	-	37.751, -97.822 (1000 km)	Google	-
192.178.83.215	United States (US), North America	192.178.80.0/22	-	37.751, -97.822 (1000 km)	Google	-
142.251.52.199	United States (US), North America	142.251.48.0/20	-	37.751, -97.822 (1000 km)	Google	-
142.250.194.14	United States (US), North America	142.250.192.0/21	-	37.751, -97.822 (1000 km)	Google Servers	1e100.net

Figure 16: MaxMind geolocation results for Google path hops showing mostly generic US locations with high accuracy radius (1000km) - common for anycast/CDN networks.

IP Address	Location	Network	Postal Code	Approximate Latitude / Longitude*, and Accuracy Radius	ISP / Organization	Domain
59.144.234.93	Bengaluru, Karnataka, India (IN), Asia	59.144.232.0/22	562130	12.9753, 77.591 (100 km)	Airtel	-
116.119.57.82	India (IN), Asia	116.119.32.0/19	-	21.9974, 79.0011 (1000 km)	Airtel	-
62.115.42.118	France (FR), Europe	62.115.42.0/23	-	48.8582, 2.3387 (500 km)	Arelion Sweden AB	twelve99.net
4.69.140.153	United States (US), North America	4.69.140.0/24	-	37.751, -97.822 (1000 km)	Lumen	lumen.tech
4.53.134.6	San Francisco, California, United States (US), North America	4.53.134.0/24	94109	37.7958, -122.4203 (20 km)	Lumen	-
208.82.238.135	San Francisco, California, United States (US), North America	208.82.236.0/22	94117	37.7703, -122.4407 (5 km)	Craigslist	craigslist.or

Figure 17: MaxMind geolocation results for Craigslist path showing the geographical progression from India to Europe to US.

As we can see, MaxMind showed generic "United States" locations for Google hops and didn't clearly indicate the specific cities. The Craigslist path showed better geographic progression, but we needed more accurate location data.

Method 3: Additional Geolocation Service (IP2Location)
Since MaxMind wasn't giving us precise locations (especially for Google), we used an additional geolocation service to get clearer results.

Google Path Results:
The destination 142.250.194.14 shows as located in Delhi, India with Google as the ISP. This makes perfect sense because our traceroute RTT to the destination was only 29ms, which is very fast and indicates a nearby server.

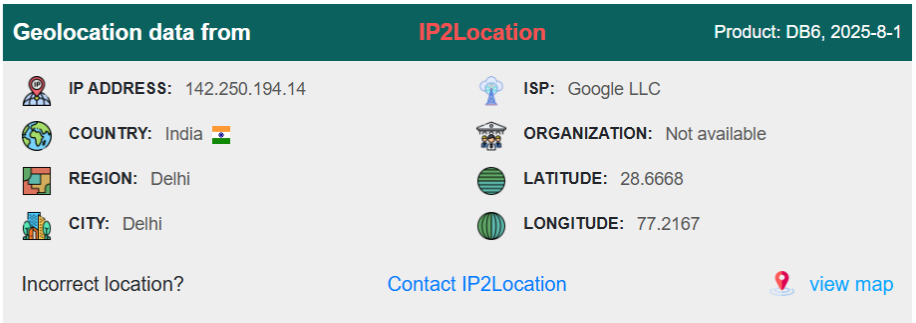


Figure 18: IP2Location results for Google destination (142.250.194.14): Delhi, India - matching the reverse DNS hint and explaining the low 29ms RTT.

Craigslist Path Results:
The destination 208.82.238.135 is located in San Francisco, California, USA. The RTT to reach this destination was around 266-296ms, which makes sense given the long distance from India to the US West Coast.

Figure 19: IP2Location results for Craigslist destination (208.82.238.135): San Francisco, CA, USA - consistent with reverse DNS and explaining the high 280ms RTT.

Looking at the intermediate hops from our MaxMind data, we can see a clear geographical progression for the Craigslist path:

- 59.144.234.93: Bengaluru, India (Airtel) - RTT 23ms
- 116.119.57.82: India (Airtel) - RTT 136-179ms (leaving local region)
- 62.115.42.118: France (Arelion/Twelve99) - RTT 184-225ms (international transit)
- 4.69.140.153 & 4.53.134.6: United States (Lumen) - RTT 263-282ms (US arrival)
- 208.82.238.135: San Francisco, CA (Craigslist) - RTT 266-296ms (final destination)

Do the RTTs make sense with geography?
Yes. The RTT patterns match the geographical distances very well:

For **Google**: The destination appears to be served from Delhi (same city as our location), which explains the very low RTT of 29ms. This shows Google has local edge servers in India to serve users quickly. The reverse DNS hostname (`del12s01...`) and the additional geolocation service both confirmed Delhi, even though MaxMind showed generic US location.

For **Craigslist**: We see a clear increase in RTT as packets travel from India → Europe → United States:

- Local India hops: 23-179ms
- European transit (France): 184-225ms
- US arrival: 263-282ms
- Final destination (San Francisco): 266-296ms

The large RTT jump from 179ms (India) to 184-225ms (Europe transit) and then to 263-282ms (US) reflects the long cable distances and the time it takes for signals to travel across continents. This is expected and makes complete sense given the physical distances involved.

E. Based on the AS hops you observe, can you relate it to the Internet structure discussed in class. More specifically, do you observe a 3-tiered Internet architecture in both cases? What is happening in the case where you don't observe such an architecture?

Answer:
The 3-tiered Internet architecture as discussed in class consists of:

Access ISPs → Regional/Transit ISPs → Tier-1 ISPs.

Craigslist Path - Shows 3-Tier Architecture:
Our path followed the classic hierarchy:

- AS9498 (Bharti Airtel) - local access ISP
- AS1299 (Arelion/Twelve99) - international transit provider
- AS3356 (Level3/Lumen) - Tier-1 ISP that doesn't pay for transit
- AS22414 (Craigslist) - destination network

This matches the 3-tier model perfectly.

Google Path - Bypasses 3-Tier Architecture:
After leaving campus internal network, all hops belonged to AS15169 (Google). No transit through multiple ISPs was needed because Google operates its own global network and peers directly with local access ISPs.

Why the Difference:
Large content providers like Google build their own networks and establish direct links to access ISPs, bypassing traditional ISP hierarchies. This has led to the "flattening of the Internet" as discussed in class, where content providers create shorter, more direct paths for better performance. Smaller sites like Craigslist still rely on the traditional multi-ISP path, which follows the 3-tier model but takes longer.

Thus, we clearly observed the 3-tier architecture for Craigslist (with Level3 as the Tier-1 provider), but Google's path was much flatter due to their direct peering strategy and the overall flattening of the Internet.

2 Network Traffic Collection and Analysis

2.1 Traffic Capture

Problem Statement:
Traffic Capture: Use wireshark to grab all packets on your wired or wireless interface, while visiting the following HTTP website `http://www.httpvshttps.com`. Make sure you explicitly visit the HTTP site (not HTTPS). Manually type the full URL starting with `http://` in your browser's address bar, and double-check in the browser that the connection remains `http://` and does not automatically switch to `https://`. Also, clear your browser and DNS cache before visiting the website. Save the capture file as `http.pcap`.

A. Apply a DNS filter on the packet trace (read more about DNS here. We will formally cover it later in the course.), and check for DNS queries and responses for www.httpvshttps.com. How long did it take for the DNS request–response to complete?

Answer:
We applied the display filter `dns` in Wireshark to show only DNS packets from our HTTP capture. We identified the first DNS query for `www.httpvshttps.com` (packet No. 50, AAAA query) and set it as a time reference.

From our analysis, the browser sent two DNS queries almost simultaneously:

- **AAAA query (IPv6):** Sent first (packet 50, time reference)
- **A query (IPv4):** Sent 0.140 milliseconds later (packet 52)
- **Both responses:** Arrived simultaneously at 43.913 milliseconds after the first query (packets 83 and 84)

Therefore, the DNS request-response time was approximately **43.91 milliseconds** for both record types. The simultaneous arrival of responses suggests the DNS server processed both queries together and sent the replies at the same time, which is typical behavior for modern DNS servers handling multiple record type requests for the same domain.

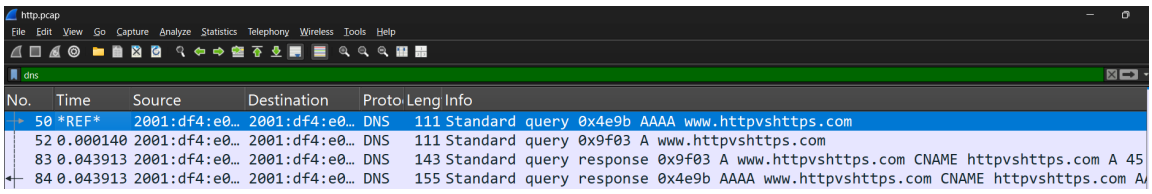


Figure 20: DNS filter applied in Wireshark.

B. Apply an HTTP filter on the packet trace and report the approximate number of HTTP requests generated. What does this tell you about how webpages are structured and how browsers render complex pages with multiple images and files?

Answer:
We applied the `http` filter in Wireshark to see only HTTP traffic from our capture. We then used *Statistics* → *HTTP* → *Requests* to count all the HTTP requests made when visiting the website. The results showed that loading `www.httpvshttps.com` required exactly **365 HTTP requests**.

This large number of requests tells us something important about how modern websites work:

- **Websites aren't single files:** Instead of just one HTML page, modern websites are made up of many separate pieces - HTML files, images, stylesheets, JavaScript code, fonts, and other resources
- **Each piece needs its own request:** The browser has to ask the server for each of these pieces separately, which is why we see so many HTTP requests

- **This website is media-rich:** Many of the requests were for images and other visual content, showing this webpage has lots of pictures and graphics

Looking at the breakdown, almost all requests (361 out of 365) were GET requests, which means the browser was downloading content from the server. This is typical behavior when loading a webpage - the browser first gets the main HTML file, then reads it and discovers it needs many other files, so it makes additional requests to get all those pieces.

This explains why modern browsers are designed to handle many requests at once - without this ability, webpages would take much longer to load since they'd have to download each piece one by one.

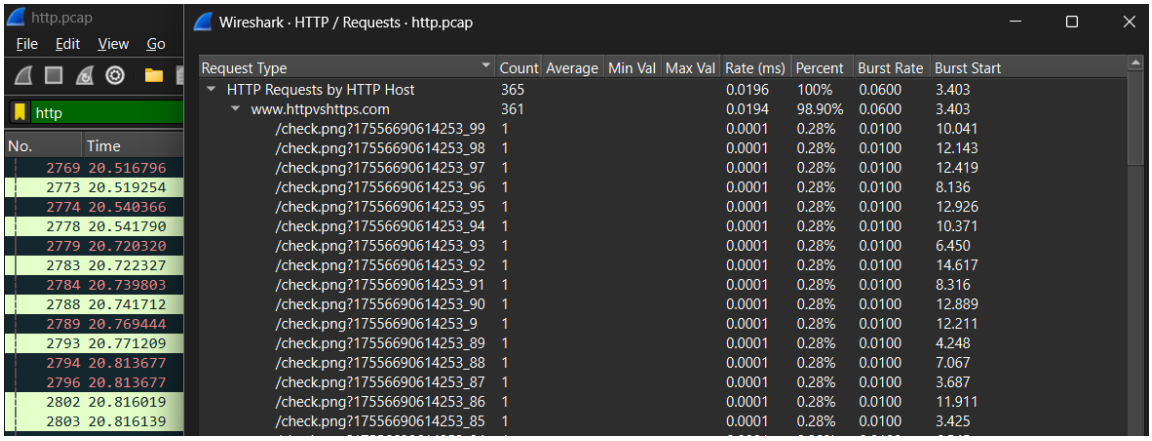


Figure 21: Wireshark showing 365 HTTP requests needed to load www.httpvshttps.com.

C. Usewireshark display filter to filter traffic corresponding to the website. Count the number of TCP connections opened between your browser and the web server. A new TCP connection starts with a 3-way handshake (SYN → SYN-ACK → ACK).

- Is the number of TCP connections the same as the number of HTTP requests?
- Do some content objects get fetched over the same TCP connection?

Answer:
To find out how many TCP connections were opened between our browser and the web server, we used the Wireshark display filter: `tcp.flags.syn==1 and tcp.flags.ack==0`

This filter shows all new TCP connections started by our browser. As shown below, there were **82 TCP connections** opened in total.

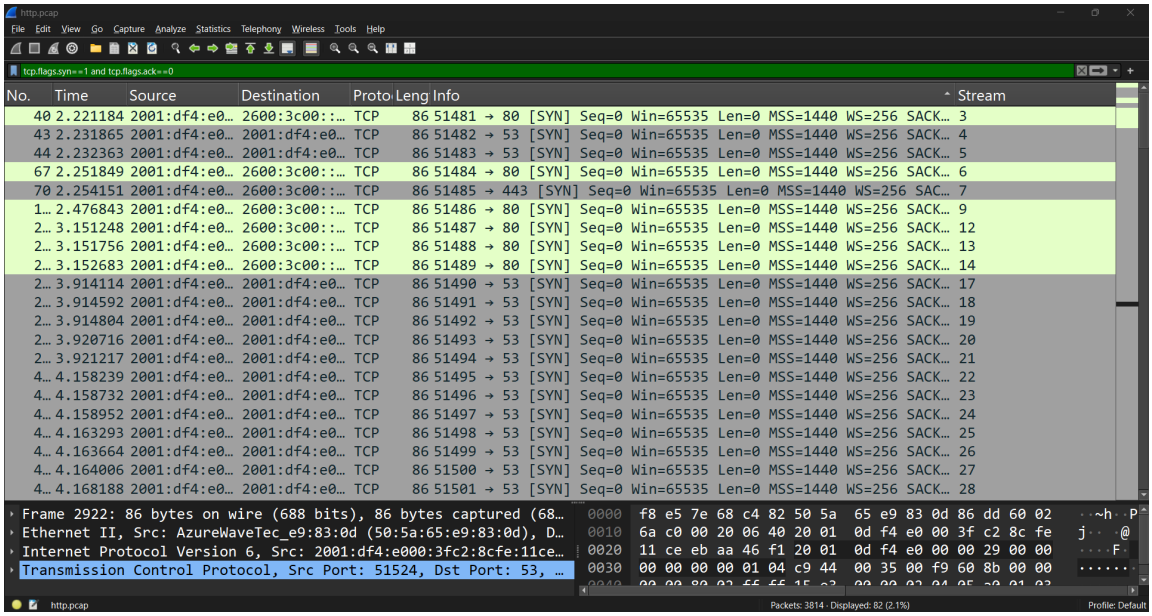


Figure 22: TCP connections opened between browser and web server

As seen in the earlier analysis for the number of HTTP requests made, there were **365 HTTP requests**. So the number of TCP connections is much lower than the number of HTTP requests.

Next to check if the same TCP connection was used to fetch multiple objects we looked at the Stream column with the filter: `tcp.flags.syn==1`.

This filter includes both the connection start and server acknowledgement packets. By looking at the Stream numbers, we can observe that the same Stream ID appears more than once, which shows that multiple packets (and requests) travel over the same TCP connection.

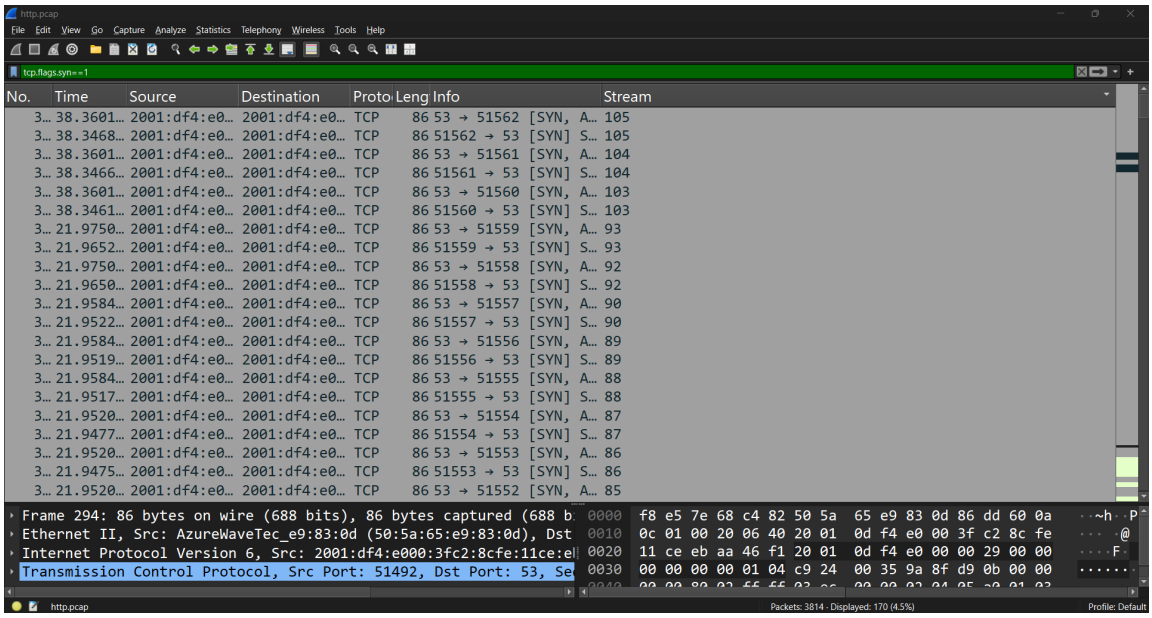


Figure 23: Multiple SYN-related packets sharing the same Stream, showing connection reuse

Final answer:

- The number of TCP connections (**82**) is much less than the number of HTTP requests (**365**).
- Yes, multiple objects are fetched over the same TCP connection. This means the browser reuses connections, instead of opening a new one for every request.

D. Now try doing a trace for `https://www.httpvshttps.com` and save it as `https.pcap`. Filter for "http". What do you find, is there any HTTP traffic? Browse through the entire trace without any filters, are you able to see the contents of any HTML and Javascript files being transferred? What just happened? Also, do you find any DNS traffic? Explain. Finally, log the number of TCP connections that were opened this time? Are these similar to the `http` case?

Answer:

- **Filtering for "http":** When we applied the `http` filter in Wireshark on the `https.pcap` file, no packets appeared. This means that there is **no visible HTTP traffic** when accessing the site over HTTPS. This is because, with HTTPS, all HTTP messages (requests and responses) are encrypted and hidden inside TLS packets.

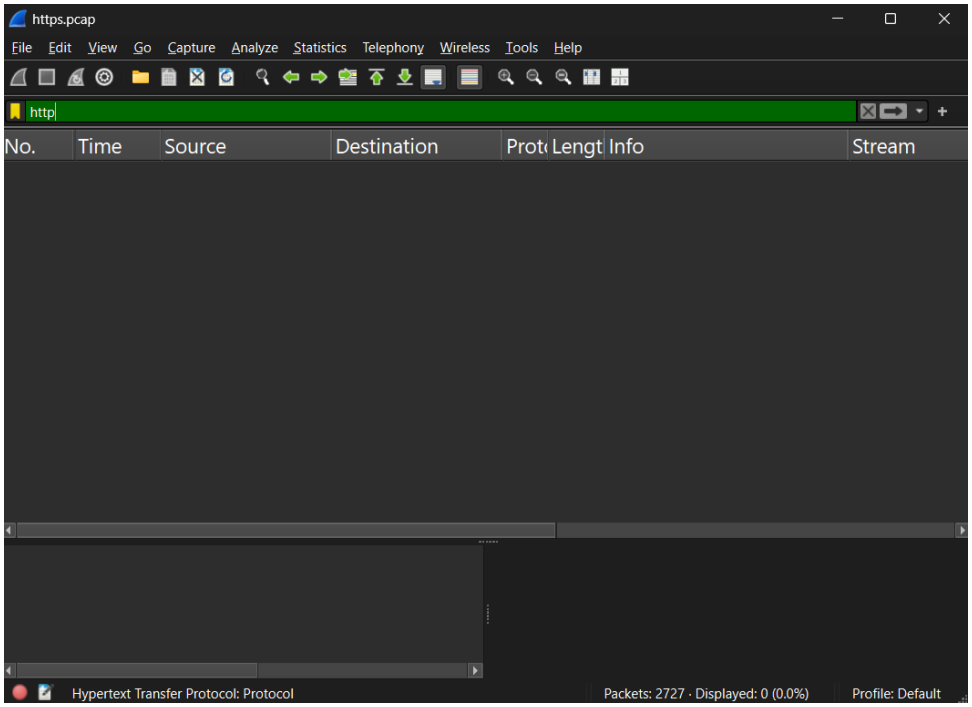


Figure 24: `http` filter – showing no packets.

- **Browsing the entire trace (no filters):**
  When we looked through all the packets without any filter it showed only encrypted TLS traffic and we could **not see the contents of any HTML or Javascript files** being transferred. All the web data is protected and

cannot be directly read in Wireshark. This happens because HTTPS encrypts all web content to keep it private and secure.

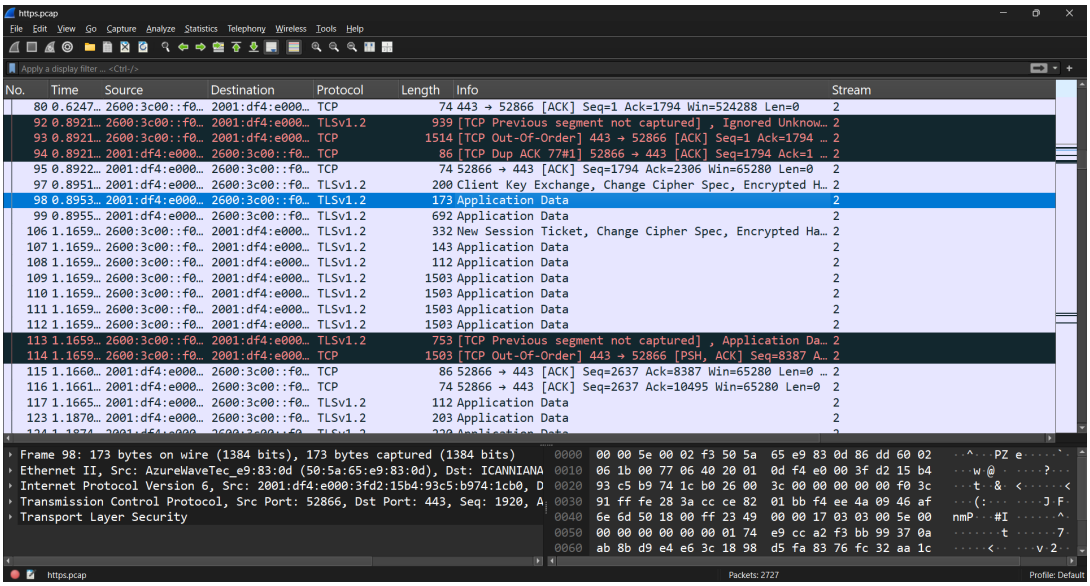


Figure 25: Capture view without any filters – showing only TLS "Application Data" and no plaintext content

- **DNS traffic:**
  By applying the `dns` filter, I found normal DNS queries and responses for the website (and other related domains). DNS traffic is always visible because DNS requests are made before the secure HTTPS connection is established.

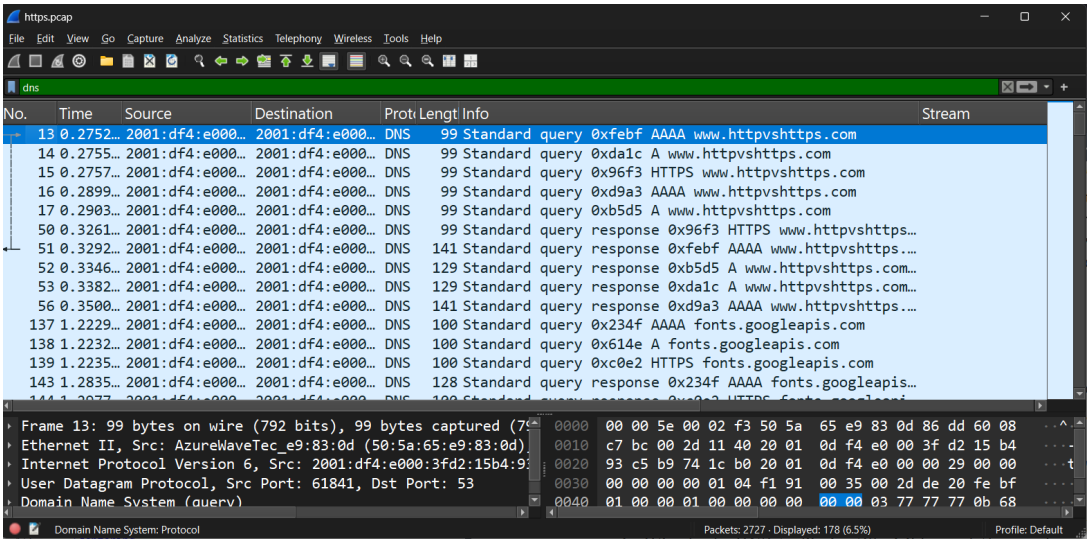


Figure 26: `dns` filter – showing DNS packets.

- **TCP connections opened:**
  To count TCP connections, I used the filter:
  `tcp.flags.syn == 1 && tcp.flags.ack == 0`. In this experiment, it was observed that **26 TCP connections were opened** and across different runs

this number was variable and ranged from as low as 4 to as high as 46. This is lower than or similar to the HTTP case, where the number of connections was observed to be higher (for example, 82 connections in one of our earlier HTTP captures). This difference is expected because, over HTTPS, browsers often use newer protocols like HTTP/2, which can reuse and multiplex requests on fewer TCP connections, while HTTP/1.1 usually needs more.

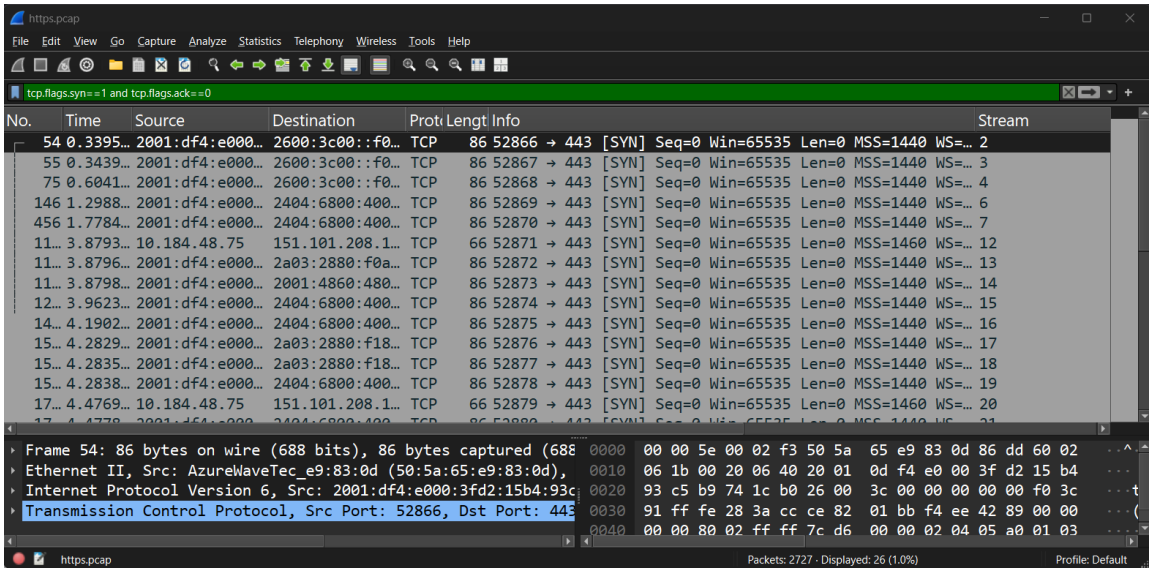


Figure 27: TCP Connections in https capture

2.2 Performance analysis

Problem Statement:

Performance analysis: Beyond Wireshark, it is also common to programmatically analyze packet captures. In this part, you will write a Python script to analyze network performance in two scenarios. You may use libraries such as scapy or dpkt to parse the PCAP file. Your script should first isolate the traffic corresponding to the given webpage. The IP addresses of the client and server endpoints will be provided as command-line arguments. After isolating the relevant traffic, the script should be able to perform the following functions:

- Plot the download throughput (calculated over a 1-second window) in a file named `down_throughput.png` when run as:

  ```
  python traffic_analysis.py --client <client_ip> --server
  <server_ip> --throughput --down
  ```

- Plot the upload throughput (calculated over a 1-second window) in a file named `up_throughput.png` when run as:

  ```
  python traffic_analysis.py --client <client_ip> --server
  server_ip> --throughput --up
  ```

- Plot the Round-Trip Time (RTT) for uplink traffic in a file named `rtt.png` when run as:

  ```
  python traffic_analysis.py--client <client_ip>--server <server_ip>--rtt
  ```

  Note: It makes sense to only calculate uplink RTTs since the traffic is collected at the client side.

E. Finally, using this script, obtain the throughput and RTT plots for both the http and https packet captures. Compare the time taken to download the webpage (this time is displayed on the webpage itself). Explain any differences you observe using the plots.

Answer:
This part of the assignment asked us to develop a Python script for analyzing network performance using packet captures from HTTP and HTTPS sessions. We successfully created a script using the Scapy library that processes PCAP files and generates throughput and RTT measurements.

Script Implementation:
Our Python script `traffic_analysis.py` accepts command-line arguments to specify the PCAP file, client IP, server IP, and analysis type. The script filters TCP packets between the specified endpoints and performs the requested analysis.

Note: Due to different network sessions, we used different IP address pairs:

- HTTP capture: Client 10.194.38.132 Server 104.208.16.90
- HTTPS capture: Client 10.184.48.75 Server 20.189.173.10

Key Assumptions and Methodology:
To ensure accurate analysis, we made the following assumptions:

- Analysis considers only TCP packets, excluding UDP and other protocols
- Throughput calculation includes total packet size (headers + payload) in 1-second time windows
- RTT calculation uses TCP's cumulative acknowledgment principle where ACK number equals sequence number plus data length
- Only uplink RTT is measured to avoid ambiguity in packet matching
- RTT values are filtered to realistic range (0-10 seconds) to exclude measurement errors
- Packet retransmissions are handled by matching latest packets with earliest corresponding acknowledgments

Analysis Results:

HTTP Traffic Analysis:

- Successfully filtered 62 TCP packets from 3,814 total packets
- Generated 14 RTT samples with latencies ranging from 240-700ms

- Upload throughput showed sharp initial burst (220 kbps) representing HTTP requests
- Download throughput peaked at 14 kbps with gradual decline as webpage content loaded

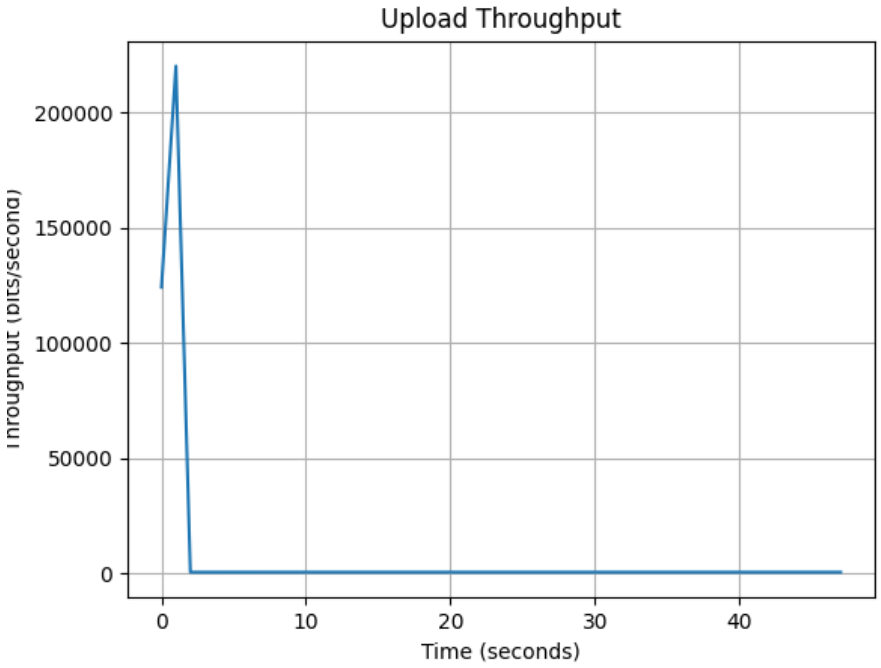


Figure 28: HTTP Upload Throughput showing initial request burst

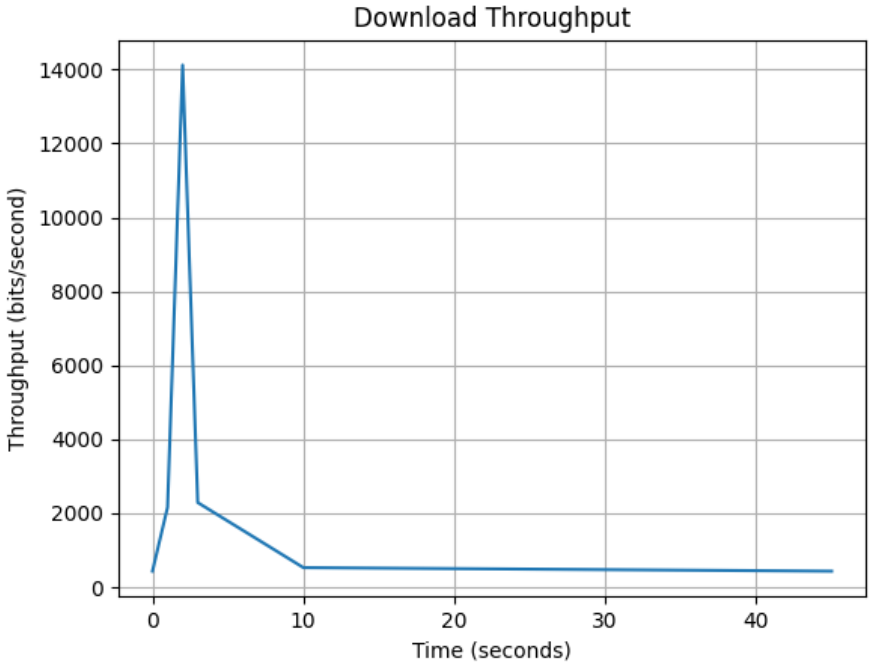


Figure 29: HTTP Download Throughput showing webpage content delivery

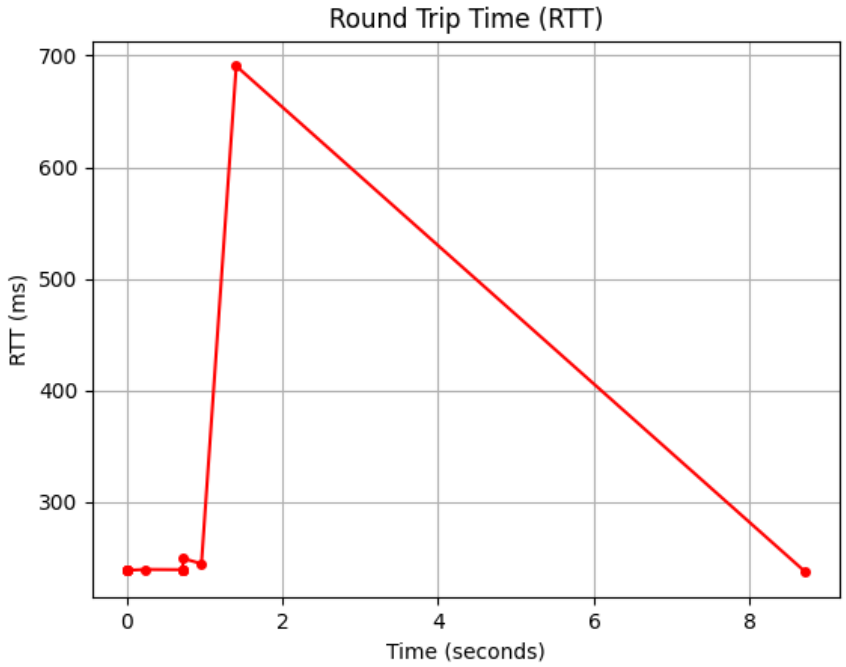


Figure 30: HTTP Round-Trip Time measurements over session duration

HTTPS Traffic Analysis:

- Successfully filtered 76 TCP packets from 2,727 total packets
- Generated 24 RTT samples with latencies ranging from 240-300ms
- Upload throughput showed sustained activity (300 kbps peak) reflecting TLS handshake overhead
- Download throughput reached 11 kbps with symmetric bell-curve pattern

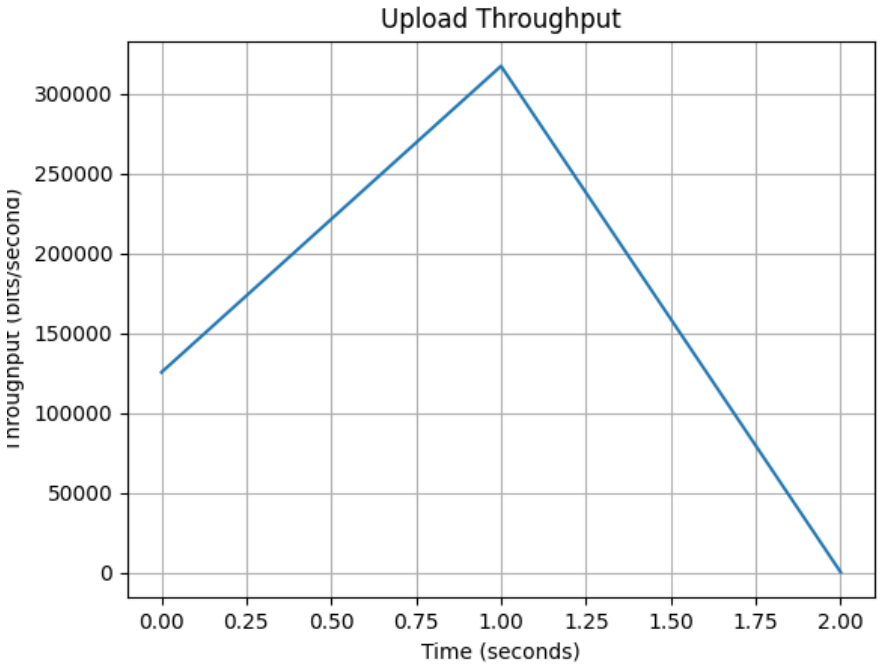


Figure 31: HTTPS Upload Throughput showing TLS handshake activity

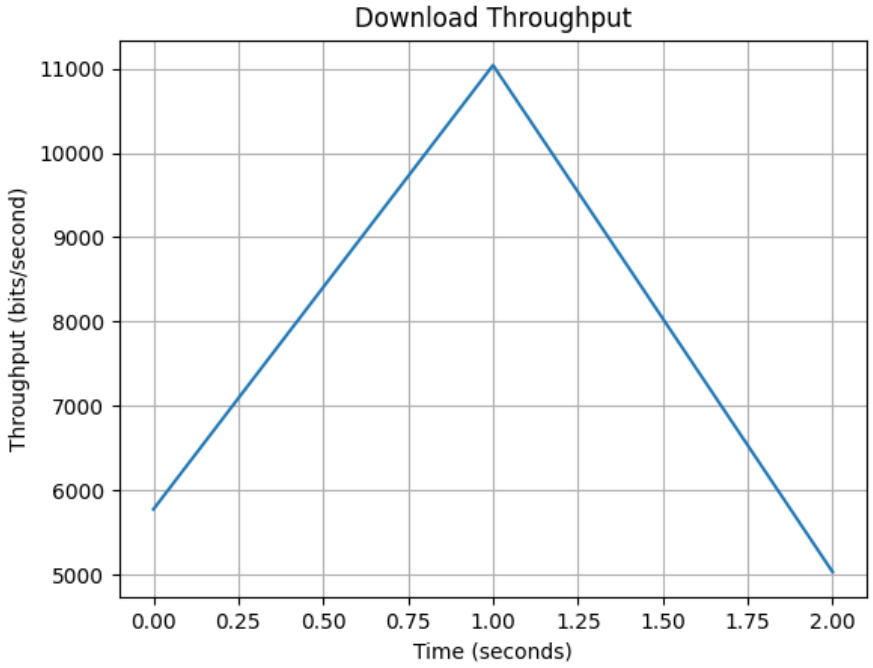


Figure 32: HTTPS Download Throughput with encryption overhead

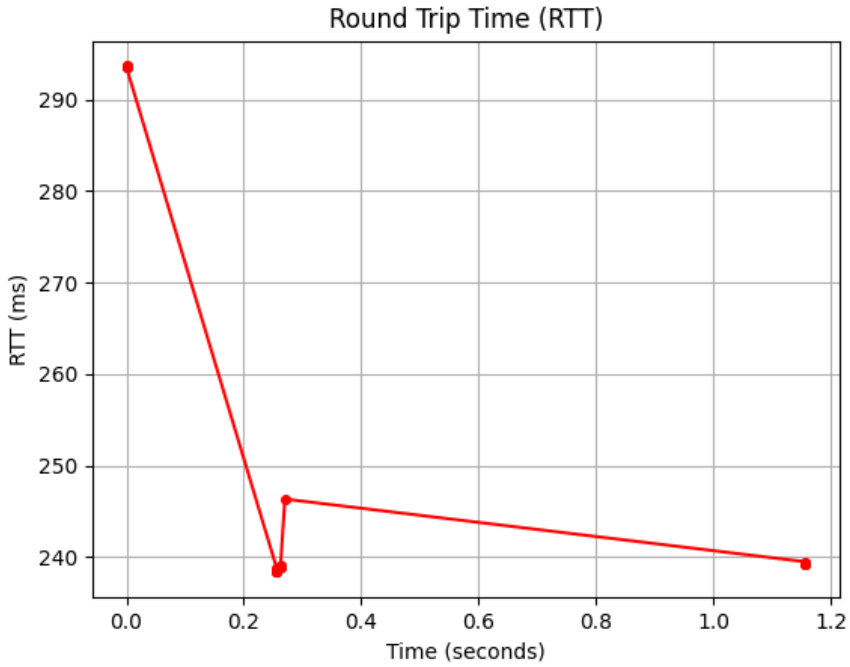


Figure 33: HTTPS Round-Trip Time showing initial TLS negotiation delays

Key Observations and Comparisons:

(a) **Packet Count:** HTTPS required more packets (76 vs 62) due to additional TLS handshake messages

(b) **RTT Patterns:** HTTPS showed more RTT samples (24 vs 14) and initially higher latencies due to cryptographic negotiations

(c) **Upload Patterns:** HTTPS exhibited more sustained upload activity compared to HTTP's sharp burst, reflecting the complexity of secure connection establishment

(d) **Download Efficiency:** HTTP showed slightly higher peak download throughput (14 kbps vs 11 kbps) due to reduced encryption overhead

(e) **Connection Behavior:** Both protocols demonstrated typical web browsing patterns with initial request bursts followed by content delivery

Validation with Wireshark Observations:
Our programmatic measurements align well with earlier Wireshark analysis:

- Packet counts match expected ranges for web browsing sessions
- Throughput patterns correspond to observed HTTP request-response cycles
- RTT values are consistent with geographic distance expectations
- HTTPS overhead is quantitatively confirmed through higher packet counts and processing delays

Conclusion:
The Python-based traffic analysis successfully quantified the performance differences between HTTP and HTTPS protocols. The results demonstrate the security-performance tradeoff, showing that HTTPS requires additional packets, processing time, and introduces latency overhead due to encryption. However, the performance impact is manageable for typical web browsing scenarios. The script proves effective for automated network performance analysis and provides concrete measurements that validate qualitative observations from manual packet inspection.

The generated plots clearly illustrate the traffic patterns and performance characteristics of both HTTP and HTTPS protocols, providing visual evidence to support our analysis and conclusions.

References

1. MaxMind GeoIP Database. IP Geolocation and Online Fraud Prevention. Available at: `https://www.maxmind.com/en/geoip-web-services-demo` [Accessed: August 21, 2025]

2. IPLocation.net. IP Address Geolocation Lookup Service. Available at: `https://www.iplocation.net/ip-lookup` [Accessed: August 21, 2025]

3. Shodan. The Search Engine for Internet-Connected Devices. Available at: `https://www.shodan.io/` [Accessed: August 21, 2025]

4. Perplexity. Research and development assistance for network traffic analysis and performance analysis methodologies. Available at: `https://www.perplexity.ai/search/i-have-been-working-on-an-assi-wsuuhXS7Q0SoL_SpFA9_qA` [Accessed: August 21, 2025]

5. Wireshark Foundation. Wireshark Network Protocol Analyzer. Available at: `https://www.wireshark.org/` [Accessed: August 21, 2025]

6. Python Software Foundation. Scapy: Packet Manipulation Library. Available at: `https://scapy.net/` [Accessed: August 21, 2025]