



Assignment 2

CS 331 - Computer Networks

Name 1: Mohit

Roll no: 23110207

Name 2: Tanishq Bhushan Chaudhari

Roll no: 23110329

Date of Submission: 26th October, 2025

[Github Repository](#)

Task: DNS Query Resolution

Basic Setup:

For this assignment, the basic setup involved downloading and using the **Mininet virtual machine (VM)**, which is freely available online. The VM was run using **Oracle VirtualBox**. To simplify interaction with the Mininet environment, I used **SSH access** from my Windows operating system.

To avoid repeatedly typing lengthy SSH commands, I created a configuration file in the Windows SSH directory. The configuration enabled quick and consistent connections to the Mininet VM by simply using the command `ssh mininet`. Additionally, to support graphical applications when needed, I configured **X11 forwarding** and installed **XLaunch** to enable GUI-based operations through SSH.

Part A

1. SSH access:

```
C:\Users\ASUS>ssh mininet
mininet@localhost's password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-42-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

New release '22.04.5 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Sun Oct 19 07:30:30 2025
mininet@mininet-vm:~$ ls
mininet  oflops  oftest  openflow  pox
```

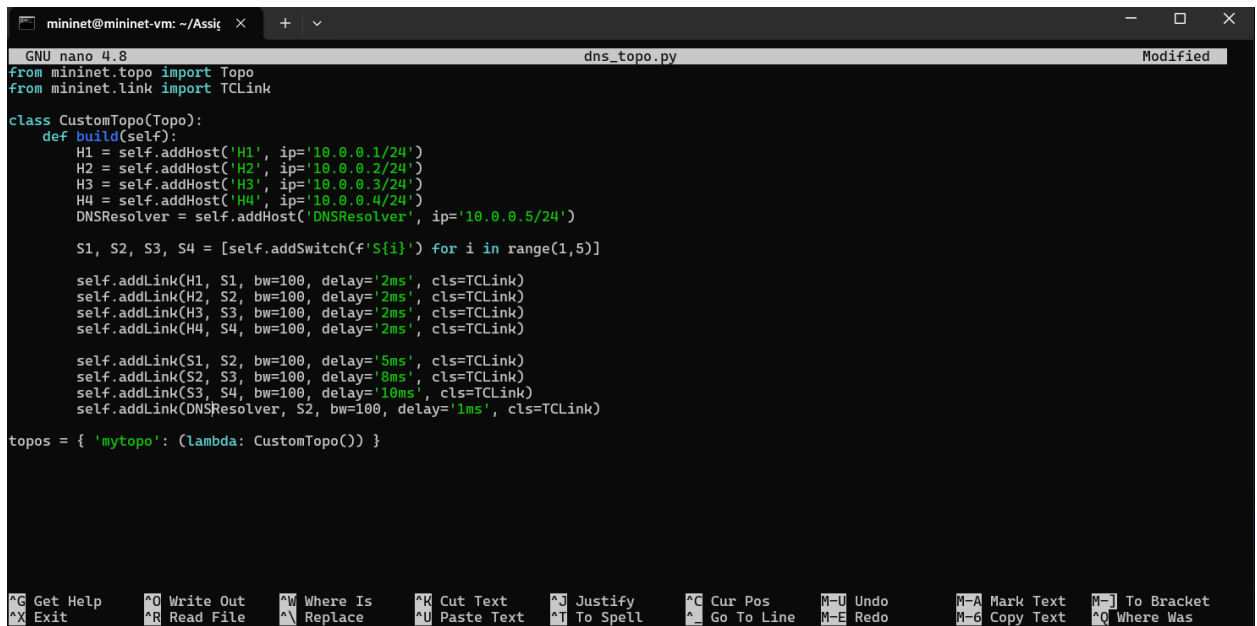
2. Creating a folder for this assignment:

```
mininet@mininet-vm:~$ mkdir Assignment2
mininet@mininet-vm:~$ ls
Assignment2  mininet  oflops  oftest  openflow  pox
mininet@mininet-vm:~$ cd Assignment2
mininet@mininet-vm:~/Assignment2$ |
```

3. Creating the required topology:

```
mininet@mininet-vm:~/Assignment2$ nano dns_topo.py
```

To create the required topology, I created a Python script, `dns_topo.py`. I did this using the above command. It opened a window to enter the file contents. Added the following content to it, saved it and exited.



```
GNU nano 4.8 dns_topo.py Modified
from mininet.topo import Topo
from mininet.link import TCLink

class CustomTopo(Topo):
    def build(self):
        H1 = self.addHost('H1', ip='10.0.0.1/24')
        H2 = self.addHost('H2', ip='10.0.0.2/24')
        H3 = self.addHost('H3', ip='10.0.0.3/24')
        H4 = self.addHost('H4', ip='10.0.0.4/24')
        DNSResolver = self.addHost('DNSResolver', ip='10.0.0.5/24')

        S1, S2, S3, S4 = [self.addSwitch(f'S{i}') for i in range(1,5)]

        self.addLink(H1, S1, bw=100, delay='2ms', cls=TCLink)
        self.addLink(H2, S2, bw=100, delay='2ms', cls=TCLink)
        self.addLink(H3, S3, bw=100, delay='2ms', cls=TCLink)
        self.addLink(H4, S4, bw=100, delay='2ms', cls=TCLink)

        self.addLink(S1, S2, bw=100, delay='5ms', cls=TCLink)
        self.addLink(S2, S3, bw=100, delay='8ms', cls=TCLink)
        self.addLink(S3, S4, bw=100, delay='10ms', cls=TCLink)
        self.addLink(DNSResolver, S2, bw=100, delay='1ms', cls=TCLink)

topos = { 'mytopo': (lambda: CustomTopo()) }
```

But when I ran this topology, there were errors because of the long length of the name `DNSResolver`, hence I edited the file and changed its name to `DNSR`.

On running this topo, we see:

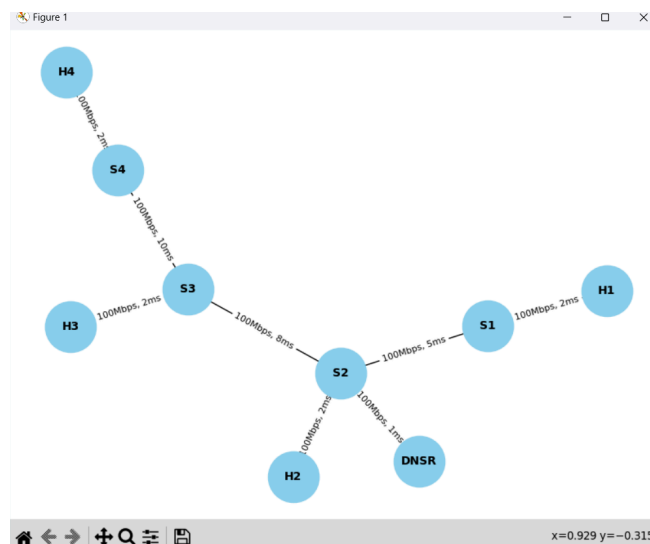
```
mininet@mininet-vm:~/Assignment2$ sudo mn --custom dns_topo.py --topo mytopo
*** Creating network
*** Adding controller
*** Adding hosts:
DNSR H1 H2 H3 H4
*** Adding switches:
S1 S2 S3 S4
*** Adding links:
(100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (DNSR, S2) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (H1, S1) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (H2, S2) (100.00Mbit 2ms delay)
(100.00Mbit 2ms delay) (H3, S3) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (H4, S4) (100.00Mbit 5ms delay) (100.00Mbit 5ms delay) (S1, S2) (100.00Mbit 8ms delay) (100.00Mbit 8ms delay)
(S2, S3) (100.00Mbit 10ms delay) (100.00Mbit 10ms delay) (S3, S4)
*** Configuring hosts
DNSR H1 H2 H3 H4
*** Starting controller
c0
*** Starting 4 switches
S1 S2 S3 S4 ... (100.00Mbit 2ms delay) (100.00Mbit 5ms delay) (100.00Mbit 1ms delay) (100.00Mbit 2ms delay) (100.00Mbit 5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 2ms delay) (100.00Mbit
8ms delay) (100.00Mbit 10ms delay) (100.00Mbit 2ms delay) (100.00Mbit 10ms delay)
*** Starting CLI:
```

To test the **successful connectivity**, I ran the `pingall` command:

```
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
DNSR -> H1 H2 H3 H4
H1 -> DNSR H2 H3 H4
H2 -> DNSR H1 H3 H4
H3 -> DNSR H1 H2 H4
H4 -> DNSR H1 H2 H3
*** Results: 0% dropped (20/20 received)
```

This image shows that all hosts successfully exchanged packets without any losses. This indicates that full network connectivity was achieved, and every host was reachable from all others as well as from the DNS resolver.

To verify that there were no errors in host naming or numerical assignments, the topology was also visualised using a Python script (`visualize_topo.py`). The script utilised libraries such as Matplotlib and NetworkX to generate a graphical representation of the network topology. On running this file, we see:



Part B

a) Extracting Query Domains

1. First, I downloaded the provided PCAP files into my Windows system.
2. Extract DNS query data using tshark:

```
PS D:\Sem 5\CN_A2\PCAPs\CS-331_PCAPs_DNS_Resolver> tshark -r "PCAP_1_H1.pcap" -Y "dns.flags.response == 0 && dns.qry.name && dns.qry.name contains '.'" -T fields -e frame.time_epoch -e dns.qry.name -e dns.flags.recdesired -e frame.len | ForEach-Object { $_ -replace "`t", "," } | Out-File "H1_queries.csv" -Encoding utf8
```

This command reads the PCAP file, extracts all DNS query packets (not responses) whose domain names contain a dot (to filter out system noise), outputs selected fields (timestamp, query name, recursion flag, and frame length), converts tabs to commas, and saves the result as a CSV file. Then the file was copied to the VM.

```
C:\Users\ASUS>scp "D:\Sem 5\CN_A2\PCAPs\CS-331_PCAPs_DNS_Resolver\H1_queries.csv" mininet:/home/mininet/Assignment2/
mininet@localhost's password:
H1_queries.csv 100% 4553 635.2KB/s 00:00
```

The same process was repeated for the remaining three capture files to ensure all necessary data was available within the VM environment.

b) Resolving and Recording Data

1. To resolve the DNS queries we extracted in the previous steps using the default host resolver, I wrote a Python script, `resolve_and_measure.py`.

```
mininet@mininet-vm:~/Assignment2$ nano resolve_and_measure_partb.py
mininet@mininet-vm:~/Assignment2$ chmod +x ~/Assignment2/resolve_and_measure_partb.py
```

This Python script measures DNS resolution performance for each host in the simulated topology. It uses `socket.getaddrinfo()` to send DNS queries sequentially for all domains in an input file and times each lookup with `time.perf_counter()`. Successful lookups are counted when `getaddrinfo()` returns a result, while failures are caught as exceptions. From this, the script calculates the number of successes and failures, the average lookup latency, and throughput in bits per second, which is computed using only the request direction and the packet sizes observed in the PCAP files. The results are then appended to `results_summary_partb.csv`, storing the summary for all hosts.

2. Starting our custom topology and enabling NAT connectivity to allow internet access for hosts.

```
mininet@mininet-vm:~/Assignment2$ sudo mn --custom dns_topo.py --topo mytopo --link tc --nat
```

3. Some Checks:

```
mininet> H1 cat /etc/resolv.conf
```

```
nameserver 127.0.0.53
options edns0 trust-ad
```

This means that the DNS requests are being sent to the IP Address 127.0.0.53, which means it's using a local DNS stub resolver on the VM, not an external DNS. To check connectivity, we ping this address from H1.

```
mininet> H1 ping -c 1 127.0.0.53
PING 127.0.0.53 (127.0.0.53) 56(84) bytes of data:
64 bytes from 127.0.0.53: icmp_seq=1 ttl=64 time=0.041 ms

--- 127.0.0.53 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.041/0.041/0.041/0.000 ms
```

This means the IP address is reachable, but when we try to look up the IP address of any website like Google, it shows that the default nameserver (127.0.0.53) doesn't work inside the Mininet namespace.

```
mininet> H1 nslookup google.com
;; connection timed out; no servers could be reached
```

4. Checking UDP Sockets:

```
mininet@mininet-vm:~/Assignment2$ sudo ss -ulpn | grep 53
UNCONN    0      0      127.0.0.0:53*lo:53      0.0.0.0:*      users:((("systemd-resolve",pid=362,fd=12))
```

The command shown above lists all active UDP sockets on port 53, which is the default port for DNS. It shows which process (like **systemd-resolve**) is currently listening for DNS queries on the Mininet VM, helping identify where the DNS service is running. But when we run this command on a mininet host, we see that no process is listening on port 53 through UDP sockets.

```
mininet> H1 sudo ss -ulpn | grep 53
mininet> |
```

This shows that the Mininet hosts don't have a local DNS resolver running, so their queries never reach any DNS service, which is why they don't receive any DNS replies.

5. To test the script, I switched the mininet hosts to use an **external DNS resolver**. (Google's Public Resolver)

```
mininet> H1 sh -c "echo 'nameserver 8.8.8.8' > /etc/resolv.conf"
```

Running the Python script to resolve queries from the PCAP_1_H1.pcap file from H1.

```
mininet> H1 python3 /home/mininet/Assignment2/resolve_and_measure.py H1 /home/mininet/Assignment2/H1_queries.csv
```

The results are being stored in the CSV file. After repeating the process with the other 3 hosts, we get the following data.

```
mininet@mininet-vm:~/Assignment2$ cat results_summary_partb.csv
Host,Total Queries,Success,Failure,Avg Latency (ms),Throughput (bps)
10.0.0.1,100,71,29,343.7973,1207.3277
10.0.0.2,100,67,33,397.6978,1136.2505
10.0.0.3,100,72,28,390.4873,1206.1275
10.0.0.4,100,73,27,359.3576,1398.4003
```

If we had run the script without switching the resolver, there would be 0 successes.

6. After discussing the issue with the instructor, it was suggested that we explore less invasive methods to resolve the queries. Following this, I attempted several approaches to initiate a system-resolve process or perform resolution using the VM's resolver through **Socat**. However, all of these methods were unsuccessful.
7. Then, I created the custom topology **dns_topo_nat.py** that creates the same hosts and switches, with a **NAT** node forwarding DNS queries from the hosts to the Mininet VM's at 10.0.0.6 through the switch S2. However, it didn't work on my VM because the VM's **systemd version** doesn't support **DNSStubListenerExtra**, so systemd-resolved ignored the setting and didn't listen on 10.0.0.6.
8. When the same file was executed on my friend's virtual machine, it successfully ran without any issues. Unlike my setup, my friend did not use the preconfigured Mininet VM; instead, he installed Mininet on a newer Linux-based virtual machine. The difference in behaviour can be attributed to the fact that his system is running a more recent version of **systemd**. Here is the data the Python script recorded:

```
(mn-venv) tejas@tejas:~/temp/Mohit_NAT$ cat results_summary_partb.csv
Host,Total Queries,Success,Failure,Avg Latency (ms),Throughput (bps)
H1,100,0,100,0.00,0.00
H1,100,71,29,54.06,2051.88
H2,100,67,33,263.02,1609.30
H3,100,72,28,234.50,1537.91
H4,100,73,27,249.86,1855.73
(mn-venv) tejas@tejas:~/temp/Mohit_NAT$
```

Note that this image shows an initial row with all failures. This happened when we forgot to

change the resolv.conf file to forward our requests to the IP address 10.0.0.6. When done so, the DNS queries started to get resolved through the VM.

Part C

The Mininet hosts can be configured to utilise the custom DNS resolver by modifying their DNS configuration files to forward all DNS queries to the IP address of the Mininet host running the custom DNS resolver process (10.0.0.5) instead of its own loopback address.

We can do this by simply using the following command for each Mininet host after we start our custom topology.

```
mininet> H4 sh -c "echo 'nameserver 10.0.0.5' > /etc/resolv.conf"
```

We can check its correctness after starting the custom resolver process in part D or by simply making a DNS listener on DNSR.

Part D

In this part of the assignment, we wrote a Python script (`dns_resolver.py`) inside the Mininet VM and ran it on the host DNSR, whose IP address is 10.0.0.5. This host acts as our custom DNS resolver. To make sure all DNS queries are sent to it, we updated the DNS configuration of all hosts in the topology to use 10.0.0.5 as their DNS server, as done in the previous part.

Our custom resolver performs iterative DNS resolution, starting from the predefined root DNS servers. It first contacts a root server to get the TLD (like `.com`, `.in`, etc.) information, then queries one of those TLD servers to obtain the authoritative servers for the specific domain, and finally queries an authoritative server to get the final IP address. If a server's reply does not include direct IP addresses (glue records), the resolver performs additional A-record lookups to find them before continuing. We chose a timeout of 2 seconds for each DNS request to make the process responsive.

For every query, the resolver logs all the required details such as the timestamp, resolution steps, servers contacted, and timing information. These logs are printed in a clear format in the terminal and are also written to a text file specified when running the script, allowing the results to be stored and reviewed later.

NOTE: We chose not to try all available servers at each step because doing so made resolution unnecessarily slow, especially for domains that were ultimately unresolvable. Even with a 2-second timeout per query, contacting multiple servers per stage quickly added up to very long delays. So we switched to using only the first available server, our resolver stays efficient and responsive while still demonstrating the complete iterative resolution process for valid domains.

```

1  import socket
2  import time
3  import sys
4  from dnslib import DNSRecord
5
6  ROOT_SERVERS = [
7      "198.41.0.4",
8      "199.9.14.201",
9      "192.33.4.12",
10 ]
11
12
13 def iterative_resolve(query_data):
14     query = DNSRecord.parse(query_data)
15     qname = str(query.q.qname)
16     log = []
17     start_time = time.time()
18     current_servers = ROOT_SERVERS
19     response = None
20     step = 0
21
22     while True:
23         step += 1
24         server = current_servers[0]
25         sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
26         sock.settimeout(2)
27
28         send_time = time.time()
29         try:
30             sock.sendto(query_data, (server, 53))
31             data, _ = sock.recvfrom(2048)
32             recv_time = time.time()
33         except socket.timeout:
34             log.append((
35                 "step": step,
36                 "mode": "Iterative",
37                 "stage": "Timeout",
38                 "server": server,
39                 "rtt": None,
40                 "response": ["No response (timeout)"]
41             ))
42             sock.close()
43             break
44
45         sock.close()
46         rtt = (recv_time - send_time) * 1000
47         resp = DNSRecord.parse(data)
48
49         if step == 1:
50             stage = "Root"
51         elif len(resp.auth) > 0 and not resp.rr:
52             stage = "TLD"
53         else:
54             stage = "Authoritative"
55
56         response_summary = []
57         records = resp.rr or resp.auth or []
58         if records:
59             for rr in records:
60                 response_summary.append((rr.rname -> (rr.rtype -> (rr.rdata)))
61             else:
62                 response_summary.append("Referral or empty response")
63
64         log.append((
65             "step": step,
66             "mode": "Iterative",
67             "stage": stage,
68             "server": server,
69             "rtt": round(rtt, 2),
70             "response": response_summary
71         ))
72
73         if resp.rr:
74             response = data
75             break
76
77     ns_ips = []
78     for rr in resp.rr:
79         if rr.rtype == 1:
80             ns_ips.append(str(rr.rdata))
81
82     if not ns_ips:
83         ns_names = [str(rr.rdata) for rr in resp.auth if rr.rtype == 2]
84         if not ns_names:
85             break
86
87     ns_ips = []
88     for ns in ns_names:
89         sub_q = DNSRecord.question(ns)
90         sub_resp, _ = iterative_resolve(bytes(sub_q.pack()))
91         if sub_resp:
92             sub_parsed = DNSRecord.parse(sub_resp)
93             for rr in sub_parsed.rr:
94                 if rr.rtype == 1:
95                     ns_ips.append(str(rr.rdata))
96
97     if not ns_ips:
98         break
99
100     current_servers = ns_ips
101
102     total_time = (time.time() - start_time) * 1000
103     return response, log, round(total_time, 2), qname
104

```

Running the Resolver

1. Run the custom topology and open separate terminals for the 4 hosts and DNSR using xterm.

```

*** Starting CLI:
mininet> xterm H1

```

2. In the DNSR's terminal, run the `dns_resolver.py` file with the text file where we want to save the logs.

```

Node: DNSR@mininet-vm

```

```

root@mininet-vm:/home/mininet/Assignment2# sudo python3 dns_resolver.py log_H1_partd.txt

```

Open the terminal for the host whose queries we will be resolving, change its configuration file and run the summary Python file we used in Part B.

```

root@mininet-vm:/home/mininet/Assignment2# python3 resolve_and_measure.py H1 H1_queries.csv

```

Upon running this file, we start DNS query requests on our custom resolver, and it starts recording logs. One instance of such a log with all the required fields except for the Cache status field is:

```

[2025-10-25 03:27:54] Query from 10.0.0.1 for i-butterfly.ru.
Step 1 | Mode: Iterative | Stage: Root | Server: 198.41.0.4 | RTT: 158.16 ms
Response:
ru. -> 2 -> a.dns.ripn.net.
ru. -> 2 -> d.dns.ripn.net.
ru. -> 2 -> f.dns.ripn.net.
ru. -> 2 -> b.dns.ripn.net.
ru. -> 2 -> e.dns.ripn.net.

Step 2 | Mode: Iterative | Stage: TLD | Server: 193.232.128.6 | RTT: 221.3 ms
Response:
I-BUTTERFLY.RU. -> 2 -> fred.ns.cloudflare.com.
I-BUTTERFLY.RU. -> 2 -> sofia.ns.cloudflare.com.

Step 3 | Mode: Iterative | Stage: Authoritative | Server: 108.162.193.113 | RTT: 51.63 ms
Response:
i-butterfly.ru. -> 1 -> 172.67.151.138
i-butterfly.ru. -> 1 -> 104.21.88.172

Total resolution time: 1150.78 ms

```

3. After this, I repeated this for the other three hosts and the following summary was recorded.

```

mininet@mininet-vm:~/Assignment2$ cat results_summary_partd.csv
Host,Total Queries,Success,Failure,Avg Latency (ms),Throughput (bps)
H1,100,61,39,4317.3813,55.7400
H2,100,59,41,4328.1449,51.6434
H3,100,67,33,4725.2583,60.7633
H4,100,70,30,4898.3939,64.9722

```

Comparisons

Here, we compare the summary results observed when resolving queries in part B and in part D.

a) **Success and Failure:**

Our custom resolver had a slightly higher number of failed queries compared to the VM's built-in resolver. This is expected because our implementation used a strict 2-second timeout per step and contacted only the first available server at each level (Root, TLD, Authoritative). In contrast, the system resolver typically retries multiple servers, uses adaptive timeouts, and maintains a local cache, which helps it recover from temporary network delays or server unavailability. Thus, the higher failure count in our resolver mainly reflects its simpler, single-path iterative logic and limited retry behaviour, rather than any fundamental error in DNS resolution.

b) **Latency:**

The average DNS lookup latency was much higher when using our custom resolver compared to the VM's built-in resolver. This is expected because our resolver performs true iterative resolution, contacting root, TLD, and authoritative servers sequentially for each query.

The systemd resolver, on the other hand, may use recursive upstream resolvers and caching, so many responses are served locally or from an intermediate cache, drastically reducing response time. The variation across hosts (e.g., H1 showing especially high latency) could be due to network differences or query mix; some domains might have required additional lookups or triggered timeouts.

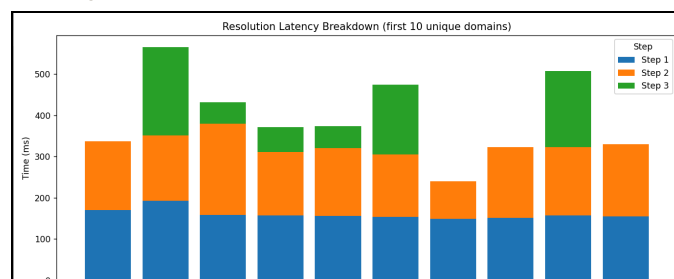
c) **Throughput:**

Throughput was also lower for our custom resolver, since throughput is inversely related to total resolution time. When queries take longer to resolve, the rate of successful lookups per second naturally drops. In contrast, the VM's built-in resolver shows higher throughput because of caching, optimised resolver logic, and parallel handling of DNS responses. One unusual observation is that Host 4 (H4) shows slightly higher throughput despite having high latency; this can happen if fewer queries failed, or if large responses were received faster than others.

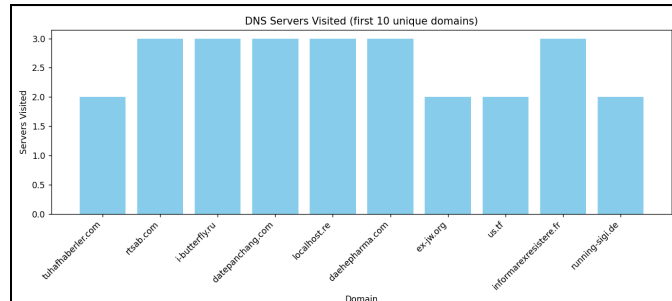
PCAP_1_H1 Analysis

1. Top 10 domains (success as well as failures included)

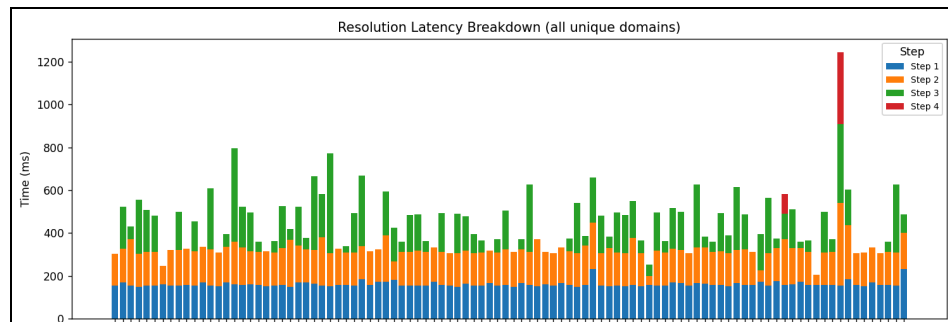
- Python script used: `analyse_all_top10_PCAP.py`
- Latency:



- No. of servers visited:



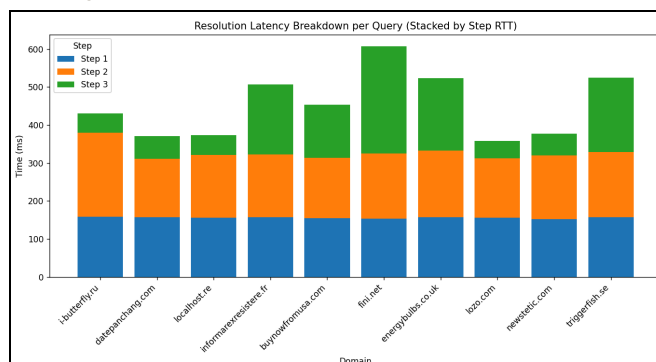
- Note that here the no. of servers visited is very low because we didn't try multiple servers for the same level to save time. Had we done that, the no. of servers visited would be quite high in some cases, specifically when DNS resolution failed. Also, on plotting the data for all queries and not only the first 10, we see that some queries also had to do a fourth step.



- To get a better understanding of latency in different steps, we can just analyse the first 10 queries that were successfully resolved.

Python script used: `analyse_success_top10_PCAP.py`

- Latency:

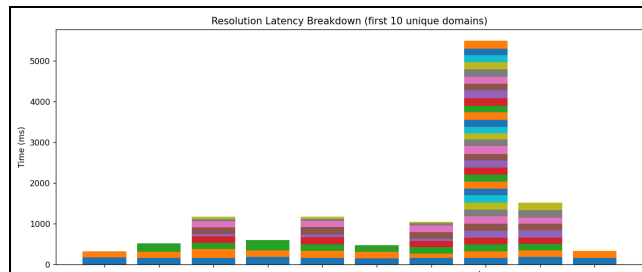


Here, the no. of steps was the same in all queries. Here we see that in step 1, latencies are very uniform because the same root server is being used for all queries. Whereas we see the most variance in step 3, probably because of different loads on authoritative servers and also due to caching of certain queries in those authoritative servers.

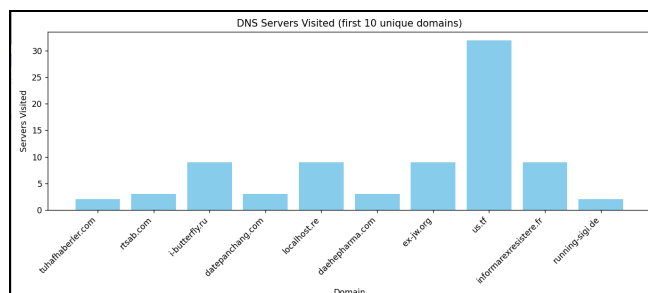
- Note that the resolver we used didn't try other servers in a step when one failed to save time. To analyse the expected functioning of a DNS resolver for the first 10 domains, we

wrote a new resolver file by the name `dns_resolver_loop.py` and saved the log for the first 10 queries in the file `log_H1_partd_loop_first10.txt`.

- Latency:



- No. of Servers visited:



- Here, the new resolver times out after 2 seconds for each server and not each step. Here, we see that when we edited the resolver to use multiple servers at each level, if one server doesn't respond in time, this may lead to a lot of failed tries to a lot of servers, leading to huge latencies in replying to DNS queries. The number of resolved queries didn't change when we used the new resolver for the first 10 queries. Both lead to 6 successes.

The successful resolution of the queries in part D also verifies the correct configuration of the mininet hosts desired in part C.

Heuristics

In this assignment, certain heuristics were applied to simplify the process, such as setting a 2-second timeout in the custom resolvers. Although a longer timeout could have been used, the number of successful responses relative to the resolution time appeared satisfactory. Even with a 2-second timeout, the process was significantly slower compared to using the default resolver. This highlights the importance of caching and illustrates how fully iterative resolution can be time-consuming. The process would have been considerably faster with caching or recursive resolution. Additionally, in the testing scripts, packet sizes were kept the same as those observed in the pcap files; however, minor variations may exist due to differences in specific DNS bits when simulated using our scripts.