CN-Assignment-2

Vedant Acharya (23110010)
Haarit Chavda (23110077)

October 26, 2025

Task: DNS query resolution

A. Simulating custom topology in Mininet

- To simulate the custom topology in Mininet shown in Figure 1, we used a Python script to add hosts (h1, h2, h3, h4, and dns), switches (s1, s2, s3, and s4), and to configure the required link bandwidth and delay between them

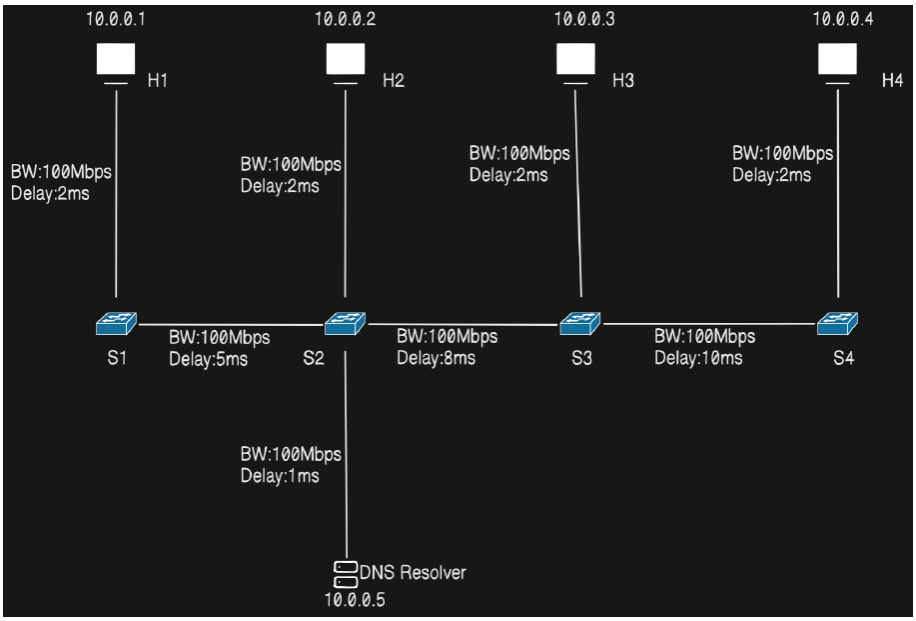

Figure 1: Custom Topology

- To verify that all host links were functioning correctly, we used the pingAll method. The output, shown in Listing 1, indicates 0% packet loss, confirming that all hosts are successfully connected according to the specified topology.

```
*** Running pingall test
*** Ping: testing ping reachability
dns -> h1 h2 h3 h4
h1 -> dns h2 h3 h4
h2 -> dns h1 h3 h4
h3 -> dns h1 h2 h4
h4 -> dns h1 h2 h3
*** Results: 0% dropped (20/20 received)
```

Listing 1: Mininet pingall output

- We also measured the average RTT between each pair of hosts using the `ping` command. The RTT is approximately twice the sum of the propagation delays of all links between the two hosts. For example, the sum of propagation delays between h1 and h4 is $2 + 5 + 8 + 10 + 2 = 27$ ms, so the RTT is roughly $2 \times 27 = 54$ ms, which is consistent with the values shown in Table 1.

- To measure TCP bandwidth, we used the `iperf` command: first, a server is started on one host, and then a client is started on the other host. The measured bandwidth is approximately 100 Mbps for most host pairs, since all links have a bandwidth of 100 Mbps. The h2-dns pair achieves the highest bandwidth of 95.4 Mbps because they are directly connected through a single switch. Similarly, the other results shown in Table 1 can be verified.

Host Pair	Avg RTT (ms)	Bandwidth (Mbps)
h1-h2	18.220	94.2
h1-h3	34.375	92.8
h1-h4	54.431	90.7
h1-dns	16.313	94.4
h2-h3	24.393	93.8
h2-h4	44.817	91.9
h2-dns	6.643	95.4
h3-h4	28.812	93.3
h3-dns	23.017	94.0
h4-dns	43.211	92.0

Table 1: Network Metrics: Average RTT and Bandwidth between Hosts

- The `ping` from h1 to h2 worked within the Mininet console, but attempting the same from h1's Bash shell did not succeed. This occurs because Mininet internally knows h2's IP address (10.0.0.2), but h1's Bash environment does not have a hostname-to-IP mapping for h2. To resolve this, we added these mappings for each host in the `/etc/hosts` file. After this update, pinging h2 from h1's Bash worked successfully. This is shown in Figure 2.

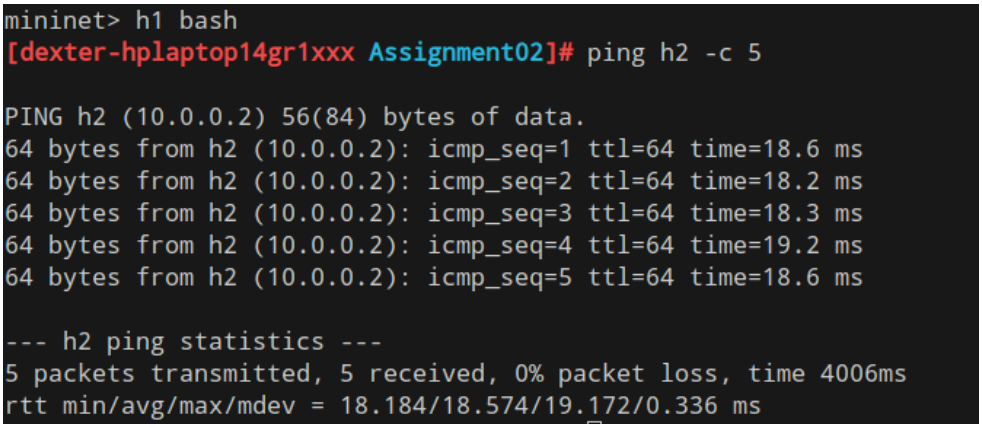

Figure 2: Ping from host's bash

B. Resolving DNS Queries from PCAP Using Default Resolver

- To enable the Mininet topology to access external networks, we configured Network Address Translation (NAT) on the `dns` host. The Google public DNS server (8.8.8.8) was set as the default resolver by adding the entry `nameserver 8.8.8.8` in each host's `/etc/resolv.conf` file. The successful ping to `google.com` from host h3's Bash shell is shown in Figure 3.

```
mininet> h3 bash
[dexter-hplaptop14gr1xxx Assignment02]# ping google.com -c 5

PING google.com (142.251.222.78) 56(84) bytes of data.
64 bytes from pnbomb-bp-in-f14.1e100.net (142.251.222.78): icmp_seq=1 ttl=112 time=206 ms
64 bytes from pnbomb-bp-in-f14.1e100.net (142.251.222.78): icmp_seq=2 ttl=112 time=229 ms
64 bytes from pnbomb-bp-in-f14.1e100.net (142.251.222.78): icmp_seq=3 ttl=112 time=149 ms
64 bytes from pnbomb-bp-in-f14.1e100.net (142.251.222.78): icmp_seq=4 ttl=112 time=172 ms
64 bytes from pnbomb-bp-in-f14.1e100.net (142.251.222.78): icmp_seq=5 ttl=112 time=56.0 ms

--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 56.030/162.521/229.283/59.912 ms
```

Figure 3: External ping from host's bash

- We used the `dpkt` library to parse the PCAP files and extract DNS queries from them. For each hostname, the corresponding DNS response was obtained using the `dig` command, as illustrated in Figure 4. In the `dig` output, the *Query time* field represents the latency of the query (in milliseconds), the *MSG SIZE rcvd* field indicates the size of the received message in bytes (which can be approximated as the total bytes transferred), and the *Status* field in the response header shows whether the query was successful. If the status is `NOERROR`, the query succeeded; if the status is `SERVFAIL` or missing, the query failed. For each host, we parsed the PCAP file and used this information to compute the average throughput (total bytes divided by latency), the average lookup latency, and the total number of successful and failed queries. The results for each host are shown in Figure 2.

```
[dexter-hplaptop14gr1xxx Assignment02]# dig +time=2 +tries=1 google.com


; <<>> DiG 9.20.13 <<>> +time=2 +tries=1 google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 14889
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             231     IN      A       142.250.192.110

;; Query time: 329 msec
;; SERVER: 8.8.8.8#53(8.8.8.8) (UDP)
;; WHEN: Sun Oct 26 17:36:29 IST 2025
;; MSG SIZE  rcvd: 55
```

Figure 4: Dig output

Hostname	Avg. Lookup Latency (ms)	Average Throughput (Bps)	Failed Queries	Successful Queries
h1	139	1414.83	24	76
h2	148	1057.65	28	72
h3	192	732.20	29	71
h4	192	639.33	24	76

Table 2: DNS Query Performance Metrics for Each Host

C. Configuring the hosts to use our DNS server

- In linux systems the nameserver/dns is configured in /etc/resolv.conf file. Therefore to configure the host, we will add 'nameserver 10.0.0.5' to /etc/resolv.conf file. We will do so by adding the following snippet while creating the topology in python.

```
info('*** Setting nameserver\n')
    for h in hosts:
        h.cmd("echo 'nameserver 10.0.0.5' > /etc/resolv.conf")
```

Listing 2: Configuring the hosts to use our dns server

- We also need to ensure that the logic written to handle the dns requests in python is running at 10.0.0.5. For that we will run server.py when we create the network topology in python. Now lets verify if we can ping google.com from host h2. **Note: Ensure that server.py on 'dns' uses port 53**

- In Figure 6 we can see that the server field is 10.0.0.5 on port 53 that uses udp protocol. This ensures that the nameserver is set successfully.

```
mininet> h2 ping google.com
PING google.com (142.250.192.46) 56(84) bytes of data.
64 bytes from 142.250.192.46: icmp_seq=1 ttl=113 time=36.8 ms
64 bytes from 142.250.192.46: icmp_seq=2 ttl=113 time=26.8 ms
64 bytes from 142.250.192.46: icmp_seq=3 ttl=113 time=27.0 ms
64 bytes from 142.250.192.46: icmp_seq=4 ttl=113 time=28.3 ms
64 bytes from 142.250.192.46: icmp_seq=5 ttl=113 time=27.0 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 13055ms
rtt min/avg/max/mdev = 26.813/29.180/36.795/3.841 ms
```

Figure 5: Ping checking google after changing the nameserver

```
mininet> h2 dig google.com

; <<>> DiG 9.18.30-0ubuntu0.24.04.2-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 40004
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             300     IN      A       142.250.192.46

;; Query time: 356 msec
;; SERVER: 10.0.0.5#53(10.0.0.5) (UDP)
;; WHEN: Sun Oct 26 18:12:42 IST 2025
;; MSG SIZE  rcvd: 44
```

Figure 6: dig checking google after changing the nameserver

D. DNS resolution for PCAPs using custom DNS

- For this, we wrote a new client.py that would extract the domain name for all DNS packets from the pcap files and send the DNS queries to our DNS server with the recursive bit set to 1. We use dpkt to parse the pcap files and dnspython to craft and send the DNS packets.
- As mentioned, we will do logging on the server side with all the required fields. In Table 3 are the first 20 entries from the log.

	Timestamp	Client IP	Domain	Mode	Server$_IP$	Step	Response type	RTT(ms)	Cache Status	Cumulative Time(ms)	Cumulative Bytes
0	2025-10-26 16:00:18	10.0.0.1	tuhafhaberler.com.	Recursive	198.41.0.4	Root	Referral	159.910200	MISS	159.910200	530
1	2025-10-26 16:00:18	10.0.0.1	tuhafhaberler.com.	Recursive	192.41.162.30	TLD	Referral	156.917600	MISS	316.827700	673
2	2025-10-26 16:00:18	10.0.0.1	tuhafhaberler.com.	Recursive	-	NaN	Failure	NaN	MISS	316.827700	673
3	2025-10-26 16:00:18	10.0.0.1	rtsab.com.	Recursive	-	Cache	Referral	NaN	HIT	0.000000	0
4	2025-10-26 16:00:19	10.0.0.1	rtsab.com.	Recursive	192.35.51.30	Authoritative	Referral	173.587600	MISS	173.587600	202
5	2025-10-26 16:00:19	10.0.0.1	rtsab.com.	Recursive	193.14.90.50	Authoritative	Referral	199.806000	MISS	373.393700	323
6	2025-10-26 16:00:19	10.0.0.1	rtsab.com.	Recursive	-	NaN	Failure	NaN	MISS	373.393700	323
7	2025-10-26 16:00:19	10.0.0.1	i-butterfly.ru.	Recursive	198.41.0.4	Root	Referral	168.025600	MISS	168.025600	376
8	2025-10-26 16:00:19	10.0.0.1	i-butterfly.ru.	Recursive	193.232.128.6	TLD	Referral	226.714500	MISS	394.740100	496
9	2025-10-26 16:00:19	10.0.0.1	sofia.ns.cloudflare.com.	Recursive	-	Cache	Referral	NaN	HIT	394.740100	496
10	2025-10-26 16:00:19	10.0.0.1	sofia.ns.cloudflare.com.	Recursive	192.35.51.30	Authoritative	Referral	168.453900	MISS	563.193900	1036
11	2025-10-26 16:00:20	10.0.0.1	sofia.ns.cloudflare.com.	Recursive	162.159.0.33	Authoritative	Response	49.446100	MISS	612.640000	1166
12	2025-10-26 16:00:20	10.0.0.1	i-butterfly.ru.	Recursive	173.245.58.223	Authoritative	Response	46.461700	MISS	659.101700	1262
13	2025-10-26 16:00:20	10.0.0.1	fred.ns.cloudflare.com.	Recursive	-	Cache	Referral	NaN	HIT	659.101700	1262
14	2025-10-26 16:00:20	10.0.0.1	fred.ns.cloudflare.com.	Recursive	162.159.4.8	Authoritative	Response	51.098600	MISS	710.200300	1390
15	2025-10-26 16:00:20	10.0.0.1	i-butterfly.ru.	Recursive	172.64.33.113	Authoritative	Response	50.654500	MISS	760.854800	1486
16	2025-10-26 16:00:20	10.0.0.1	datepanchang.com.	Recursive	-	Cache	Referral	NaN	HIT	0.000000	0
17	2025-10-26 16:00:20	10.0.0.1	datepanchang.com.	Recursive	192.35.51.30	Authoritative	Referral	175.729900	MISS	175.729900	390
18	2025-10-26 16:00:20	10.0.0.1	datepanchang.com.	Recursive	108.162.194.214	Authoritative	Response	396.848200	MISS	572.578100	474
19	2025-10-26 16:00:21	10.0.0.1	localhost.re.	Recursive	198.41.0.4	Root	Referral	151.883100	MISS	151.883100	250

Table 3: DNS Query Log Summary

- The plots for the first 10 URLs showing the total number of DNS contacted and latencies are in Figure 7 and Figure 8, respectively.
- Note that the first time .com is queried number of external DNS servers contacted is higher than at later times. This is due to caching.

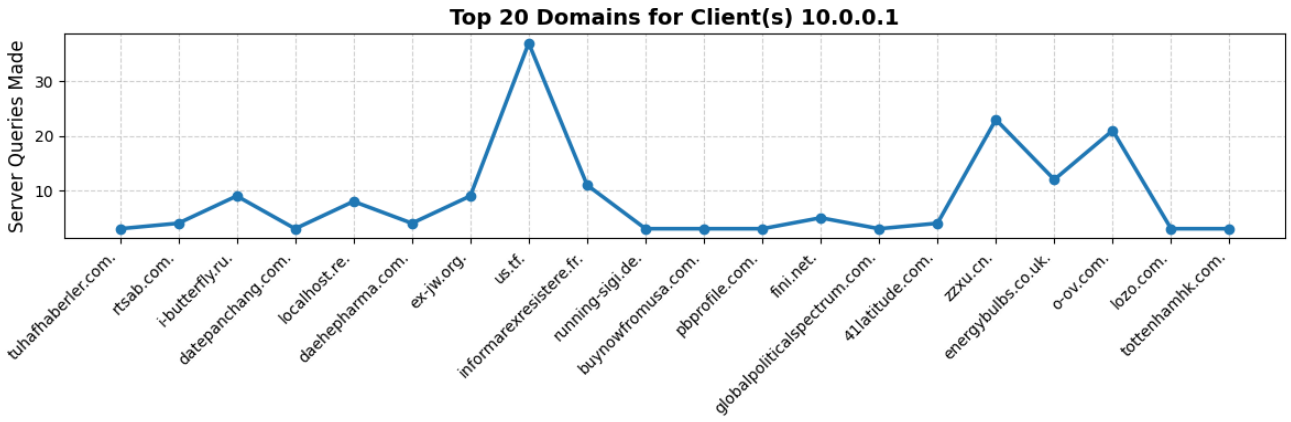

Figure 7: external dns servers contacted per query

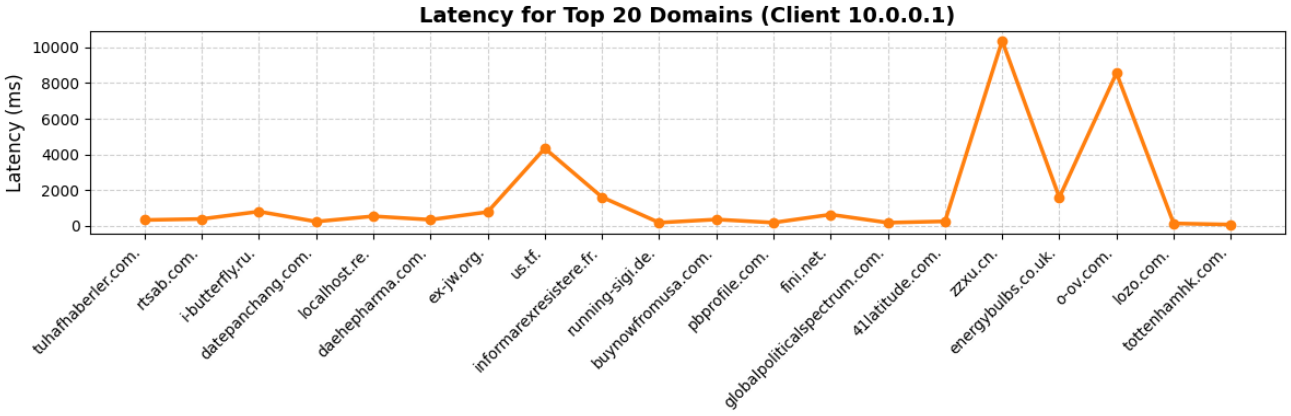

Figure 8: Latencies per queries

- The main difference between the default resolver and our custom resolver is performance, our custom resolver is approximately ten times slower. This slowdown primarily occurs because our resolver builds the DNS responses from scratch, and the cache is initially empty. In contrast, widely used public resolvers like 8.8.8.8 benefit from substantial caching due to their heavy traffic and repeated queries.
- Additionally, our custom resolver currently processes requests sequentially, as it does not yet support multithreading. This limitation prevents it from handling multiple queries concurrently, further contributing to the latency overhead.
- Overall, the combination of an unpopulated cache and lack of parallelism leads to significantly higher response times in our custom DNS resolver.

Hostname	Avg. Lookup Latency (ms)	Average Throughput (Bps)	Failed Queries	Successful Queries
h1	1357.65	1014.78	30	70
h2	944	926.93	33	67
h3	1680.68	660.47	28	72
h4	1173.42	940.82	28	72

Table 4: DNS Query Performance Metrics for Each Host Using custom DNS

E. Implementing recursive mode

- We implemented a DNS resolver that supports recursive mode. This server mainly consists of four functions: `lookup`, `lookup_recurse`, `lookup_additional`, and `lookup_authority`.
- First, the domain name query sent to the server enters the `lookup` function, where we traverse the root servers present in the list. We used three root name servers: `198.41.0.4`, `199.9.14.201`, and `192.33.4.12`. Then, we call `lookup_recurse` with the root IP, which sends the query to the given IP using the `dnspython` library.
- If the query results in a referral (i.e., the answer field is empty), and an additional field is present in the response, we traverse the additional field to obtain the IP addresses of the TLD servers by calling `lookup_additional`. This function again calls `lookup_recurse` with the TLD IP as an argument. The same process repeats if the TLD response does not contain an answer, in which case a query is made to the authoritative server.
- If no additional field is present, the authority field is traversed using `lookup_authority`. This field does not contain direct IPs of referrals but has the corresponding NS (name servers). Therefore, we make a separate lookup query for this NS to obtain its IP address. After retrieving the NS IP, we send the main query to this IP again using `lookup_recurse`. Our server's behaviour is shown in Figure 9.

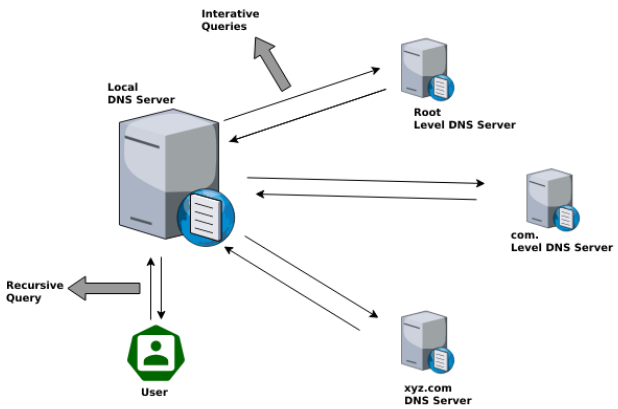

Figure 9: Recursive DNS

F. Implementing cache

- Caching is primarily handled by the `update_cache` function, which updates a dictionary named `dns_cache` whenever a reference to the `additional` field is found. It traverses the list of IPs in the `additional` section and stores the last IP address. This update process takes place inside the `lookup_recurse` function.
- Another level of caching is implemented when the final answer is obtained. This is stored in a dictionary named `response_cache`, which is maintained within `dns_cache`. Whenever the server receives a DNS query, it first checks whether the corresponding entry exists in `response_cache`. If it is present, a cache hit occurs; otherwise, the `lookup` function is called.
- During lookup, the resolver also checks for each domain name component (after every dot) to see if it already exists in the cache. If it does, the query begins from that cached IP instead of starting again from the root server, thereby reducing lookup time.
- Overall, the results of part E and F are shown in the table 5.

Hostname	**Hit Rate** (%)	**Avg. Lookup Latency (ms)**	**Avg. Throughput (Bps)**	**Failed Queries**	**Successful Queries**
h1	26.37	1357.65	1014.78	30	70
h2	29.82	944.00	926.93	33	67
h3	29.52	1680.68	660.47	28	72
h4	30.46	1173.42	940.82	28	72

Table 5: DNS Query Performance Metrics for Each Host Using Custom DNS Resolver

References

- Recursive DNS Resolver – GitHub Repository
- GFG - What is recursive DNS?