Computer Networks Assignment – 2

Rishabh Jogani (23110276) | Shardul Junagade (23110297)

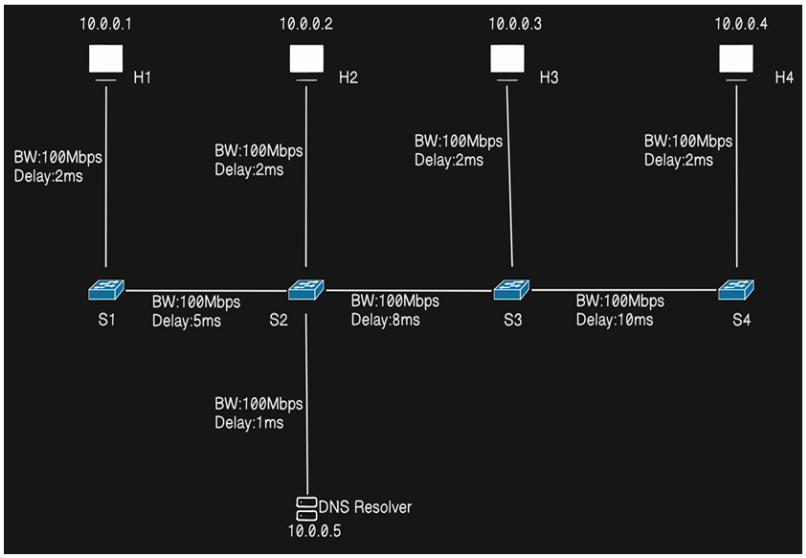
Repository Link: ShardulJunagade/cn-assignment-2

DNS Query Resolution

We set up the Mininet environment inside a virtual machine running on VMware. The VM was configured with Ubuntu, and Mininet was installed along with the required networking tools like tshark and python and its libraries. To make the workflow smoother, we connected the VM to VS Code using an SSH extension. This allowed us to edit, run, and monitor all our Python scripts and topology files directly from VS Code while the Mininet processes executed inside the virtual machine.

The foundation of this assignment is a custom Mininet topology. It consists of 4 hosts (h1-h4), 4 switches (s1-s4) in a line, and a dedicated host named dns (10.0.0.5) connected to switch s2. All links use TCLink to simulate real-world bandwidth and delay that were mentioned in the assignment instructions.

Following is the topology diagram from the assignment:



```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.log import setLogLevel

class ImageTopo(Topo):
    """
    - 4 switches in a line
    - 4 hosts, one per switch
    - 1 DNS server connected to S2
    """
    def build(self):
        # --- Add Hosts ---
        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')
        dns_host = self.addHost('dns', ip='10.0.0.5/24')


        # --- Add Switches ---
        s1 = self.addSwitch('s1', failMode='standalone')
        s2 = self.addSwitch('s2', failMode='standalone')
        s3 = self.addSwitch('s3', failMode='standalone')
        s4 = self.addSwitch('s4', failMode='standalone')

        # --- Add Links ---
        # Host to Switch links
        self.addLink(h1, s1, bw=100, delay='2ms')
        self.addLink(h2, s2, bw=100, delay='2ms')
        self.addLink(h3, s3, bw=100, delay='2ms')
        self.addLink(h4, s4, bw=100, delay='2ms')
        self.addLink(dns_host, s2, bw=100, delay='1ms') # Link the 'dns' host

        # Switch to Switch links
        self.addLink(s1, s2, bw=100, delay='5ms')
        self.addLink(s2, s3, bw=100, delay='8ms')
        self.addLink(s3, s4, bw=100, delay='10ms')

if __name__ == '__main__':
    setLogLevel('info')

    topo = ImageTopo()
```

The `topology.py` file was modified to include net.addNAT().configDefault(). This connects the Mininet network to the host VM's internet connection to gain access to the Internet.

```
    net = Mininet(topo=topo,
                  link=TCLink,
                  controller=None)

    # --- ADD NAT FOR INTERNET ACCESS (TASK B) ---
    net.addNAT().configDefault()
    # -------------------------------------------

    net.start()

    print("Starting Mininet CLI...")
    print("Your topology is running!")
    CLI(net)

    print("Stopping network...")
    net.stop()
```

Task A: Topology Simulation and Connectivity

After starting the simulation with 'sudo python3 topology.py', the Mininet CLI becomes available. We use three commands to verify the topology:

• nodes: To list all available nodes (hosts, switches, and the NAT node).

• net: To display the links between all nodes, verifying the physical connections.

• pingall: To test connectivity between all host-pairs

Result for nodes:

```
mininet> nodes
available nodes are:
dns h1 h2 h3 h4 nat0 s1 s2 s3 s4
```

Result for net:

```
mininet> net
dns dns-eth0:s2-eth2
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
h4 h4-eth0:s4-eth1
nat0 nat0-eth0:s1-eth3
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth3 s1-eth3:nat0-eth0
s2 lo:  s2-eth1:h2-eth0 s2-eth2:dns-eth0 s2-eth3:s1-eth2 s2-eth4:s3-eth2
s3 lo:  s3-eth1:h3-eth0 s3-eth2:s2-eth4 s3-eth3:s4-eth2
s4 lo:  s4-eth1:h4-eth0 s4-eth2:s3-eth3
mininet>
```

Result for pingall:

```
mininet> pingall
*** Ping: testing ping reachability
dns -> h1 h2 h3 h4 nat0
h1 -> dns h2 h3 h4 nat0
h2 -> dns h1 h3 h4 nat0
h3 -> dns h1 h2 h4 nat0
h4 -> dns h1 h2 h3 nat0
nat0 -> dns h1 h2 h3 h4
*** Results: 0% dropped (30/30 received)
mininet>
```

Task B: Default Resolver Performance

This task measures the DNS lookup performance of each host against the default system resolver (in this case, a public DNS server) using queries extracted from PCAP files.

Methodology:

Extracting Queries

```
extract() {
  local in_file=$1
  local out_file=$2
  echo "Extracting from ${in_file} -> ${out_file}"
  tshark -r "${in_file}" -Y 'udp.port == 53 && dns.flags.response == 0' \
    -T fields \
    -e frame.time_relative \
    -e dns.qry.name \
    -e dns.flags.recdesired \
    -e frame.len \
    -E header=y -E separator=, -E quote=d -E occurrence=f > "${out_file}"
}

extract "${PCAP_DIR}/PCAP_1_H1.pcap" "${OUT_DIR}/h1_queries.csv"
extract "${PCAP_DIR}/PCAP_2_H2.pcap" "${OUT_DIR}/h2_queries.csv"
extract "${PCAP_DIR}/PCAP_3_H3.pcap" "${OUT_DIR}/h3_queries.csv"
extract "${PCAP_DIR}/PCAP_4_H4.pcap" "${OUT_DIR}/h4_queries.csv"
```

This is the command to extract the queries from pcap files. I have written a bash file to extract queries from all pcap files together in `extract_queries.sh`. Initially, I wasnt using udp.port==53 in the above tshark command, due to which we were getting some wrong entries with their domains as “wpad” which took a lot of our time and effort, but the updated tshark command works as expected and gives 100 entries from each of the 4 pcap files. These extraced csv files were saved in `pcap_queries` folder.

Thereafter, we use the file name `resolve_with_default_dns.py` to resolve the obtained domains. We resolve the obtained domains and calculate metrics for the same in this python script.

Following metrics were obtained.

Label	Total Queries	Success Count	Failure Count	Avg Latency (ms)	Avg Throughput (qps)	Total Bytes	Throughput (bps)
h1_default.csv	100	71	29	139.719	5.432	5289.0	2298.592
h2_default.csv	100	68	32	108.937	8.194	4939.0	3237.438
h3_default.csv	100	72	28	103.561	8.604	5287.0	3639.259
h4_default.csv	100	73	27	135.517	8.352	5440.0	3634.804

Task C: Modifying DNS Configuration of Mininet Hosts

To connect to the custom DNs resolver (10.0.0.5), we can change the /etc/resolv.conf file

Then, running a DNS server process on the custom resolver on 10.0.0.5 will allow us to use tools with our custom resolver.

Following command is used to connect to custom server at 10.0.0.5

```
mininet> h1 echo "nameserver 10.0.0.5" > /etc/resolv.conf
mininet> h1 cat /etc/resolv.conf
nameserver 10.0.0.5
```

This was verified using ping and nslookup as follows:

Command: h1 ping –c 4 8.8.8.8

```
mininet> h1 ping -c 4 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=127 time=30.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=127 time=29.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=127 time=30.2 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=127 time=29.0 ms

--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 29.041/29.716/30.319/0.558 ms
```

Command: h1 nslookup google.com (after running the custom DNS resolver)

```
mininet> h1 sh -c "echo 'nameserver 10.0.0.5' > /etc/resolv.conf"
mininet> dns sh -c "python3 custom_dns_resolver.py > dns.log 2>&1 &"
mininet> h1 nslookup google.com
Server:         10.0.0.5
Address:        10.0.0.5#53

Non-authoritative answer:
Name:   google.com
Address: 192.41.162.30
Name:   google.com
Address: 192.41.162.30
```

Implementation and Functioning of the same is shown in the following part D.

Task D: DNS Resolution using Custom Resolver (10.0.0.5)

In this task, we repeated the DNS resolution experiment from Task B, but this time every host in the Mininet topology was configured to use our custom DNS resolver running on `10.0.0.5`.

Setup and Configuration

We began by redirecting all Mininet hosts (h1–h4) to use our own resolver.
The following commands were executed inside the Mininet CLI:

```
h1 sh -c "echo 'nameserver 10.0.0.5' > /etc/resolv.conf"
h2 sh -c "echo 'nameserver 10.0.0.5' > /etc/resolv.conf"
h3 sh -c "echo 'nameserver 10.0.0.5' > /etc/resolv.conf"
h4 sh -c "echo 'nameserver 10.0.0.5' > /etc/resolv.conf"
```

This ensured that all DNS queries from the hosts were routed to our custom resolver node (`dns`).
We then launched the custom resolver in the background on the DNS node using:

```
dns sh -c "python3 custom_dns_resolver.py > dns.log 2>&1 &"
```

Then, we executed the following command on host `h1` to resolve its respective list of domains extracted from the PCAP file:

```
h1 python3 resolve_with_default_dns.py --input
pcap_queries/h1_queries.csv --output results/h1_custom.csv
```

The same command was run for all four hosts with their respective CSVs.

Custom Resolver Implementation

Our resolver (`custom_dns_resolver.py`) was implemented entirely from scratch using raw UDP sockets.
It worked iteratively, starting from a predefined list of root servers, then querying the TLD and authoritative servers step-by-step until an A record was obtained.

For every query, we logged:

- Timestamp
- Domain name queried
- Resolution mode (Iterative)
- DNS server IP contacted
- Step of resolution (Root / TLD / Authoritative / Cache)
- Response or referral received
- Round-trip time for each contacted server
- Total resolution time
- Cache status (HIT / MISS)

Each log entry was written to `dns_server.log`, allowing us to track and analyze every step of the lookup process in detail. Few examples are shown as follows:

```
2025-10-28T16:37:33.290085,google.com,Iterative,10.0.0.1,Root,192.41.162.30,148.35ms,MISS
2025-10-28T16:37:33.455352,google.com,Iterative,10.0.0.1,Cache,192.41.162.30,0.00ms,HIT
2025-10-28T16:37:45.615455,telemetry.individual.githubcopilot.com,Iterative,10.0.0.254,Root,192.41.162.30,142.96ms,MISS
2025-10-28T16:37:56.863044,mobile.events.data.microsoft.com,Iterative,10.0.0.254,Root,192.41.162.30,143.87ms,MISS
2025-10-28T16:38:01.172252,mobile.events.data.microsoft.com,Iterative,10.0.0.254,Cache,192.41.162.30,0.00ms,HIT
2025-10-28T16:38:09.154346,telemetry.individual.githubcopilot.com,Iterative,10.0.0.254,Cache,192.41.162.30,0.00ms,HIT
2025-10-28T16:38:16.299503,tuhafhaberler.com,Iterative,10.0.0.1,Root,192.41.162.30,156.08ms,MISS
2025-10-28T16:38:16.472645,rtsab.com,Iterative,10.0.0.1,Root,192.41.162.30,136.73ms,MISS
2025-10-28T16:38:16.626286,i-butterfly.ru,Iterative,10.0.0.1,Root,193.232.128.6,140.07ms,MISS
2025-10-28T16:38:16.783800,datepanchang.com,Iterative,10.0.0.1,Root,192.41.162.30,153.99ms,MISS
2025-10-28T16:38:16.954787,localhost.re,Iterative,10.0.0.1,Root,194.0.9.1,137.92ms,MISS
2025-10-28T16:38:17.109629,daehepharma.com,Iterative,10.0.0.1,Root,192.41.162.30,141.93ms,MISS
2025-10-28T16:38:17.268132,ex-jw.org,Iterative,10.0.0.1,Root,199.249.112.1,150.62ms,MISS
```

Client-side results, including per-query latency and status, were saved in `results/h{i}_custom.csv`.

A consolidated performance comparison table was also saved in `results/summary_comparison.csv` which is diplsyed below:

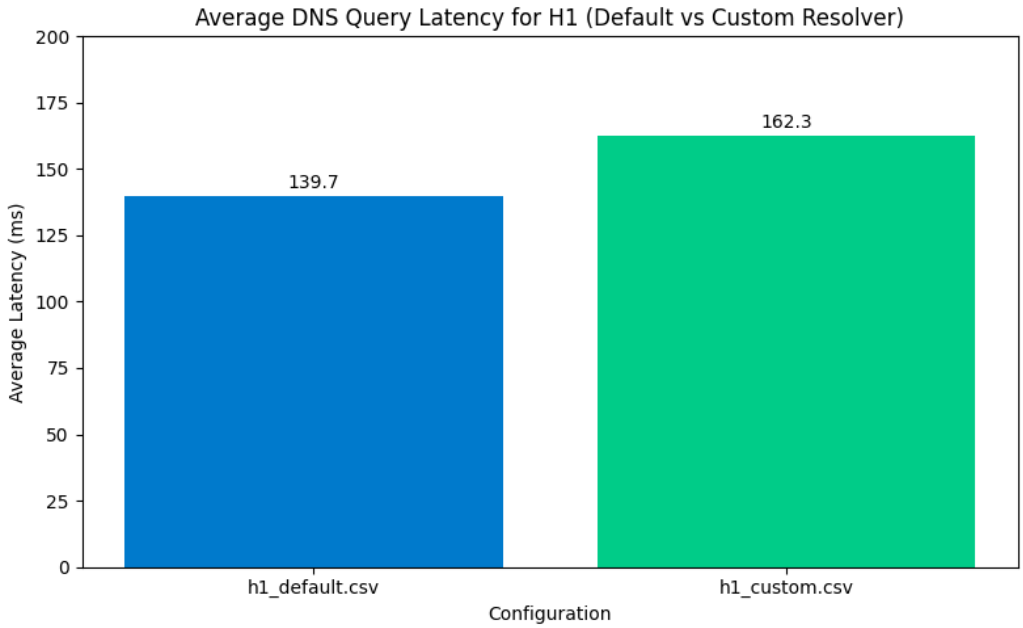
	Label	Total Queries	Success Count	Failure Count	Avg Latency (ms)	Avg Throughput (qps)	Total Bytes	Throughput (bps)
2	h1_default.csv	100	71	29	139.719	5.432	5289.0	2298.592
3	h2_default.csv	100	68	32	108.937	8.194	4939.0	3237.438
4	h3_default.csv	100	72	28	103.561	8.604	5287.0	3639.259
5	h4_default.csv	100	73	27	135.517	8.352	5440.0	3634.804
6	h1_custom.csv	100	99	1	162.346	5.237	7380.0	3091.753
7	h2_custom.csv	100	100	0	153.383	6.518	7314.0	3814.033
8	h3_custom.csv	100	100	0	167.367	5.974	7438.0	3554.592
9	h4_custom.csv	100	100	0	189.557	5.275	7470.0	3152.059

Results and Observations

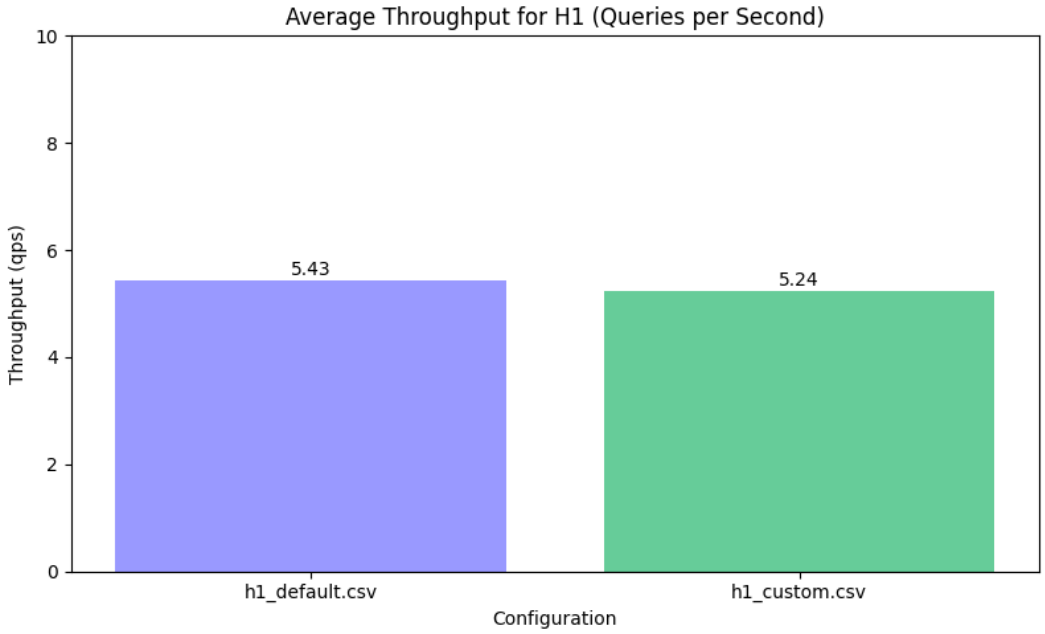
All hosts successfully resolved almost every query through our custom DNS server. Compared to the default resolver from Task B, our resolver achieved a much higher success rate but the average latency was also higher.

For further analysis, we plotted the first 10 queries from PCAP_1_H1, comparing:

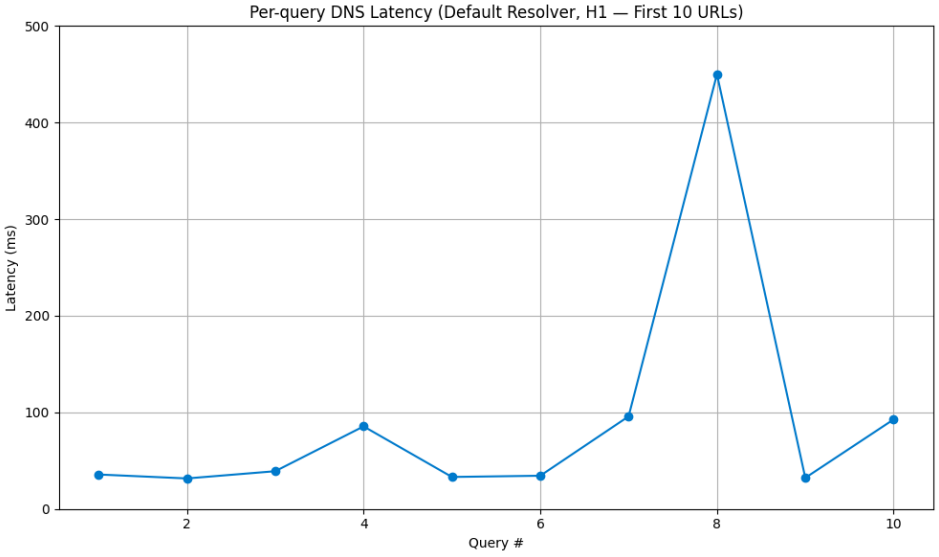
- Average DNS query latency betweent the default and custom DNS resolver



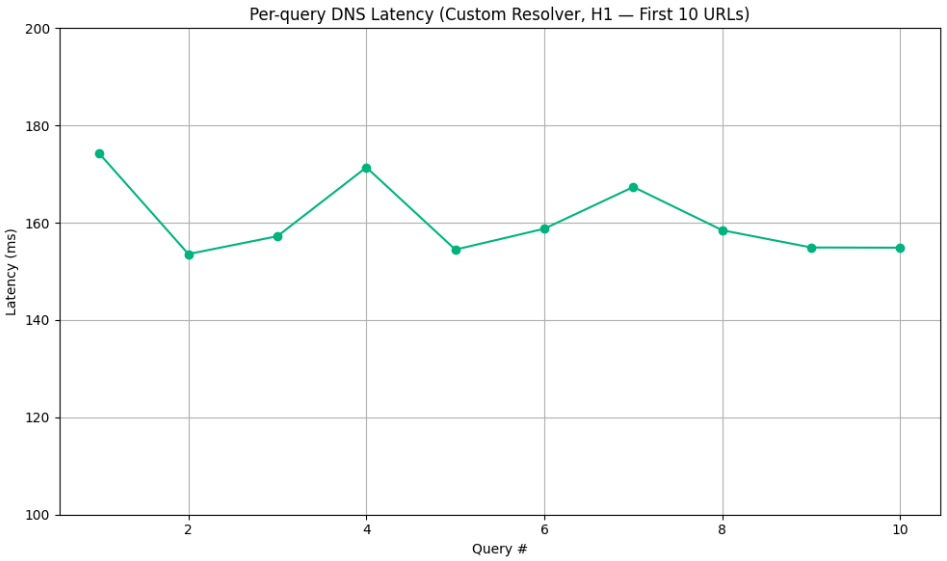
- Average throughput (in queries per second) betweent the default and custom DNS resolver



- Per-query DNS Latency for first 10 queries for default resolver



- Per-query DNS Latency for first 10 queries for custom-made DNS resolver



More plots can be found in the notebook `plots.ipynb` present in the GitHub repository.

References

1. Mininet: An Instant Virtual Network on Your Laptop (or Other PC) - Mininet
2. tshark Documentation
3. socket — Low-level networking interface — Python 3.14.0 documentation