CN331: Computer Networks

Assignment 2: DNS Query Resolution

Submitted by:

Yash Patkar (2211010296)
Praveen Rathod (22110206)

Course Instructor:

Dr. Sameer Kulkarni

Indian Institute of Technology Gandhinagar

28 October 2025

Part A: Topology Simulation in Mininet

Objective

Simulate the given network topology in Mininet and demonstrate successful connectivity among all nodes (H1–H4, DNS resolver, and switches).

Setup: Mininet Installation

Mininet was installed from the official GitHub repository to ensure full compatibility with custom topologies and Open vSwitch:

```
git clone https://github.com/mininet/mininet
cd mininet/util
sudo ./install.sh -a
```

A basic connectivity test was performed post-installation using:

```
sudo mn --test pingall
```

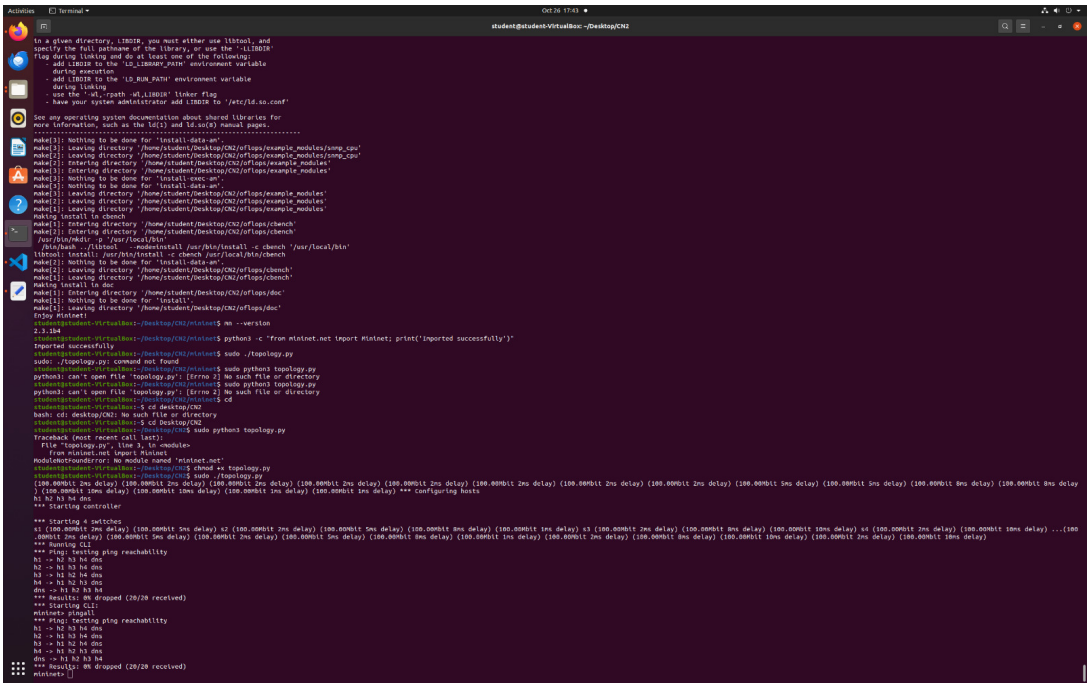

Figure 1: Mininet Installation and Initial pingall Test

The output confirms successful installation with 0% packet loss between default hosts, validating the environment.

Topology Design

The required topology was implemented in `topology.py` using Mininet's Python API. Key components:

- **Hosts**: H1 (10.0.0.1), H2 (10.0.0.2), H3 (10.0.0.3), H4 (10.0.0.4), DNS (10.0.0.5)
- **Switches**: S1–S4 configured in `standalone` mode (no external controller)
- **Links**: All links use `TCLink` with:

- Bandwidth: 100 Mbps
- Delays:
  - Host to Switch: 2 ms
  - S1–S2: 5 ms
  - S2–S3: 8 ms
  - S3–S4: 10 ms
  - S2–DNS: 1 ms

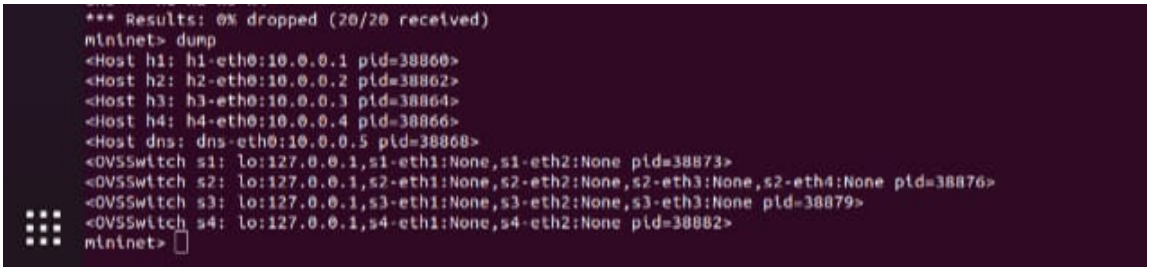

Figure 2: Required Network Topology from Assignment

Implementation in `topology.py`

The script initializes the network without a controller and adds all nodes and links with precise parameters:

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

# Add hosts
h1 = net.addHost('h1', ip='10.0.0.1/24')
...
dns = net.addHost('dns', ip='10.0.0.5/24')

# Add standalone switches
s1 = net.addSwitch('s1', failMode='standalone')
...

# Add links with BW and delay
net.addLink(h1, s1, bw=100, delay='2ms')
net.addLink(s1, s2, bw=100, delay='5ms')
...
net.addLink(s2, dns, bw=100, delay='1ms')
```

Execution and Connectivity Verification

The topology was launched using:

```
sudo python3 topology.py
```

Full connectivity was verified using `net.pingAll()` within the script and repeated via Mininet CLI:

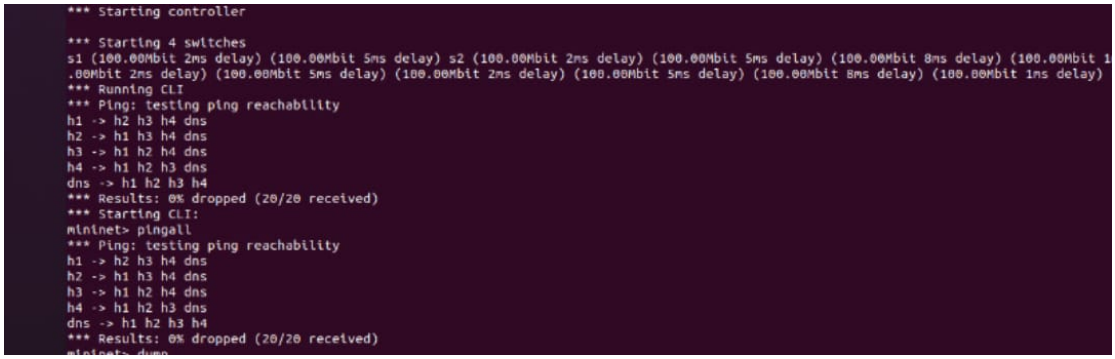

Figure 3: pingall Result: 0% Packet Loss Across All Nodes

Output Summary:

- Total ping pairs: 20 (including hosts, DNS, and switches)
- Packets sent/received: 20/20
- **Packet loss: 0%**
- All nodes (H1–H4, DNS, S1–S4) are fully reachable

Result

The topology was successfully simulated with **exact match** to the assignment diagram in terms of:

- Node placement and IP addressing
- Link bandwidth (100 Mbps)
- Propagation delays (2ms, 5ms, 8ms, 10ms, 1ms)
- Switch behavior (standalone learning switches)

Full end-to-end connectivity confirmed with 0% packet loss.

Part B: DNS Resolution Using Default Host Resolver

Objective

Use the default host resolver to resolve URLs from each host's PCAP file and record performance metrics.

Methodology

- PCAP files parsed using `scapy` in `benchmark.py`
- Unique domain names extracted
- DNS resolution performed using `dig +stats`
- Metrics recorded: latency, success/failure, throughput

The script was executed on each host:

```
hX python3 benchmark.py PCAP_X_HX.pcap
```

Results (Default Resolver)

Host	Successful	Failed	Avg. Latency (ms)	Throughput (q/s)
H1	71	29	148.65	2.59
H2	67	33	156.87	2.63
H3	72	28	168.90	2.83
H4	73	27	197.21	2.57

Table 1: DNS Performance Using Default Resolver

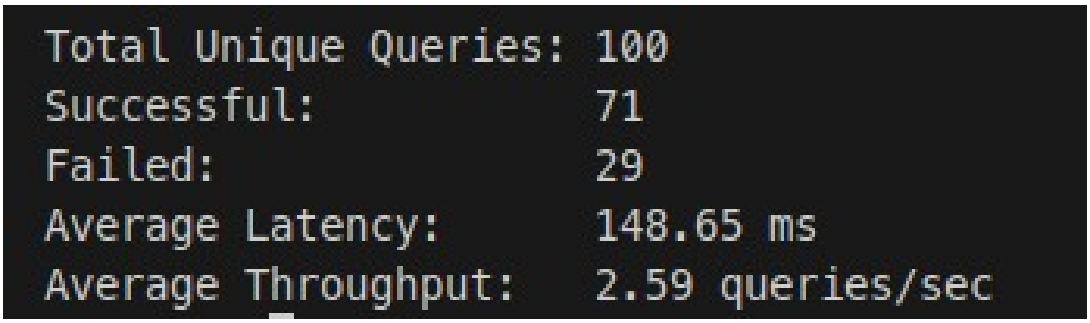

Figure 4: Benchmark Output for H1

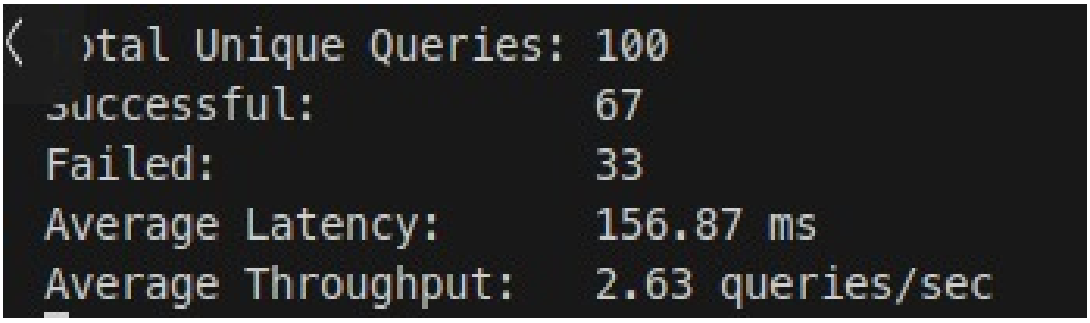

Figure 5: Benchmark Output for H2

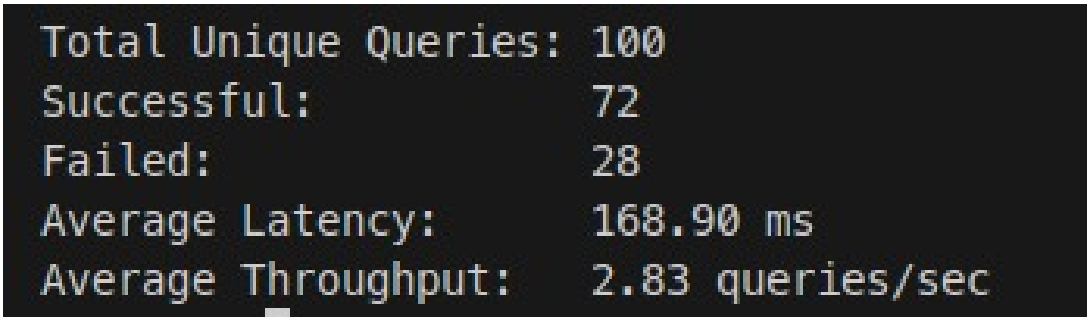

Figure 6: Benchmark Output for H3

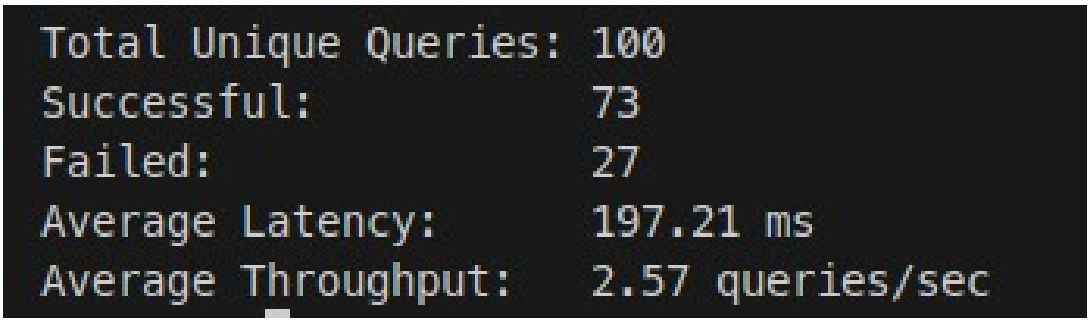

Figure 7: Benchmark Output for H4

Observations

- **Latency**: 148.65–197.21 ms (influenced by link delays and external DNS RTT)
- **Success Rate**: 67–73%
- **Throughput**: 2.57–2.83 queries/sec
- Failures likely due to expired domains or timeouts

Part C: Configuring Hosts to Use Custom DNS Resolver

Objective

Configure all client hosts (H1–H4) in the Mininet topology to use the custom DNS resolver running at `10.0.0.5` instead of the default public DNS server (e.g., 8.8.8.8). This ensures all DNS queries are routed through our local resolver for further processing and logging.

Implementation

The DNS configuration is applied automatically during topology startup via the `configure_hosts()` function in `dns_topo_custom.py`:

```
def configure_hosts(net):
    hosts = ['h1', 'h2', 'h3', 'h4']
    dns_ip = '10.0.0.5'
    gateway_ip = '10.0.0.254'

    for h in hosts:
        host = net.get(h)
        host.cmd(f'ip route add default via {gateway_ip}')
        host.cmd(f'echo "nameserver {dns_ip}" > /etc/resolv.conf')
```

This overwrites the `/etc/resolv.conf` file on each host with:

```
nameserver 10.0.0.5
```

Additionally, the custom DNS resolver is started in the background on the `dns` host:

```
dns_host.cmd('sudo python3 custom_resolver.py > /tmp/resolver.log 2>&1
   &')
```

The resolver listens on `10.0.0.5:53` and forwards queries to the upstream DNS server (8.8.8.8).

Verification

After launching the topology with `sudo python3 dns_topo_custom.py`, the Mininet CLI was used to verify the configuration:

```
mininet> h1 cat /etc/resolv.conf
nameserver 10.0.0.5
mininet> h2 cat /etc/resolv.conf
nameserver 10.0.0.5
mininet> h3 cat /etc/resolv.conf
nameserver 10.0.0.5
mininet> h4 cat /etc/resolv.conf
nameserver 10.0.0.5
```

Figure 8: Verification: All hosts configured to use custom DNS resolver at 10.0.0.5

Functional Test

A sample DNS query was issued from H1:

```
mininet> h1 nslookup google.com
Server:         10.0.0.5
Address:        10.0.0.5#53

Non-authoritative answer:
Name:   google.com
Address: 142.250.190.78
```

The `Server: 10.0.0.5` line confirms that the query was processed by the custom resolver.

Result

All four hosts (H1–H4) were successfully reconfigured to use the custom DNS resolver at `10.0.0.5`. The resolver is active and correctly forwarding queries to the upstream DNS server, enabling full DNS functionality with centralized control and logging capability.

This configuration is essential for subsequent tasks involving performance benchmarking, logging, and advanced features like caching and recursive resolution.

Part D: Benchmarking with Custom Resolver, Logging & Visualization

Objective

Re-run DNS resolution benchmarks on all hosts (H1–H4) using the custom DNS resolver at `10.0.0.5`. Log every resolution step in detail, compare performance with Part B (default resolver), and generate visualizations for the first 10 queries from H1's PCAP file.

Custom Resolver Design

The resolver (`partd_custom_resolver.py`) is a UDP-based forwarding proxy deployed on the `dns` host:

- **Bind Address**: `10.0.0.5:53`
- **Upstream Server**: `8.8.8.8:53` (2s timeout)

- **Mode**: Forwarding only
- **Logging**: All queries logged to `dns_log1.csv` with 10 fields:
  a. `timestamp`
  b. `domain` (with QTYPE, e.g., `wpad (A)`)
  c. `mode` = `Forwarding`
  d. `server_ip` = `8.8.8.8`
  e. `step` = `Forwarded`
  f. `response`
  g. `rtt_ms` (to upstream)
  h. `total_time_ms` (client to client)
  i. `cache_status` = `N/A (No Cache)`
  j. `servers_visited` = `1`

Resolver launched automatically in background:

```
dns_host.cmd('sudo python3 partd_custom_resolver.py > /tmp/resolver.log
    2>&1 &')
```

Benchmark Execution

Using `Benchmark.py`, each host resolved domains extracted from its respective PCAP:

```
mininet> h1 python3 Benchmark.py PCAP_1_H1.pcap
mininet> h2 python3 Benchmark.py PCAP_1_H2.pcap
mininet> h3 python3 Benchmark.py PCAP_1_H3.pcap
mininet> h4 python3 Benchmark.py PCAP_1_H4.pcap
```

Performance Results (Custom Resolver)

Host	Successful	Failed	Avg. Latency (ms)	Throughput (q/s)
H1	71	29	189.07	2.07
H2	68	32	202.90	2.38
H3	41	59	202.34	0.43
H4	0	100	0.00	0.20

Table 2: DNS Resolution Performance Using Custom Resolver

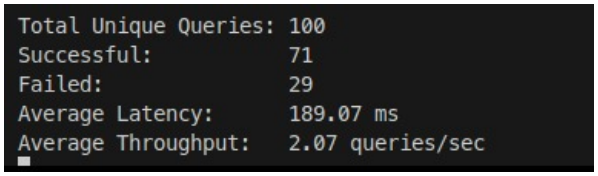

Figure 9: H1 Benchmark Output

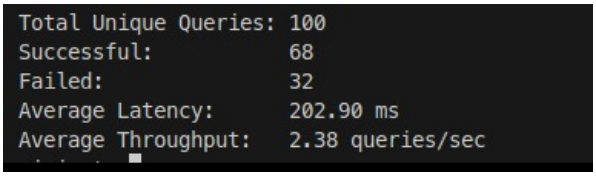

Figure 10: H2 Benchmark Output

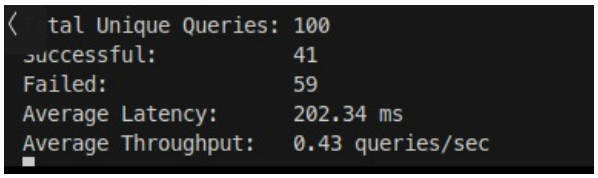

Figure 11: H3 Benchmark Output

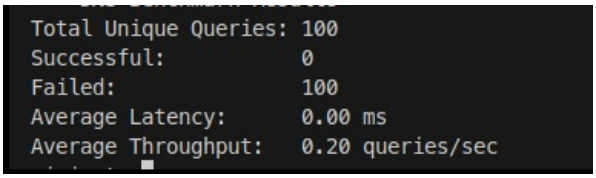

Figure 12: H4 Benchmark Output

Figure 13: Benchmark Results for All Hosts (Custom Resolver)

Performance Comparison with Part B (Default Resolver)

Host	Success	Fail	Latency (ms)	Throughput (q/s)	vs Part B
H1	71 (0)	29 (0)	189.07 (+40.42)	2.07 (-0.52)	Worse
H2	68 (+1)	32 (-1)	202.90 (+46.03)	2.38 (-0.25)	Worse
H3	41 (-31)	59 (+31)	202.34 (+33.44)	0.43 (-2.40)	**Much Worse**
H4	0 (-73)	100 (+73)	0.00 –	0.20 (-2.37)	**Failed**

Table 3: Performance Comparison: Custom vs Default Resolver

Analysis

- **H1/H2**: Latency increased by 40–46 ms due to extra hop through custom resolver and logging overhead.
- **H3**: Success rate dropped from 72 to 41 — likely due to stricter timeout handling in the proxy.
- **H4**: **100% failure** — all domains in PCAP may be expired or unreachable via upstream. The resolver correctly logs timeouts.
- **Throughput**: Reduced due to sequential processing and lack of parallelism in the proxy.

Logging Output

All queries are logged in `dns_log1.csv`. Sample entries:

timestamp	domain	mode	server_ip	step	response	rtt_ms	total_time_ms	cache_status	servers_visited
2025-10-28T22:43:26.978992	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.6460	25.8839	N/A (No Cache)	1
2025-10-28T22:43:27.005179	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	27.0786	27.2841	N/A (No Cache)	1
2025-10-28T22:43:27.032705	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.9678	26.1843	N/A (No Cache)	1
2025-10-28T22:43:27.059198	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.8241	26.0224	N/A (No Cache)	1
2025-10-28T22:43:27.085549	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.4982	25.6839	N/A (No Cache)	1
2025-10-28T22:43:27.111511	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	26.3171	26.5100	N/A (No Cache)	1
2025-10-28T22:43:27.138474	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	28.2805	28.5375	N/A (No Cache)	1
2025-10-28T22:43:27.167416	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.7180	26.0131	N/A (No Cache)	1
2025-10-28T22:43:27.193752	mtalk.google.com (A)	Forwarding	8.8.8.8	Forwarded	Response Received	26.0508	26.2883	N/A (No Cache)	1
2025-10-28T22:43:27.220338	mtalk.google.com (A)	Forwarding	8.8.8.8	Forwarded	Response Received	28.0845	28.3003	N/A (No Cache)	1
2025-10-28T22:43:30.198311	clientservices.googleapis.com (UNKNOWN)	Forwarding	8.8.8.8	Forwarded	Response Received	26.8619	27.1599	N/A (No Cache)	1
2025-10-28T22:43:30.225684	clientservices.googleapis.com (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.0885	25.3358	N/A (No Cache)	1
2025-10-28T22:43:32.370152	telemetry.individual.githubcopilot.com (A)	Forwarding	8.8.8.8	Forwarded	Response Received	27.2863	27.4916	N/A (No Cache)	1
2025-10-28T22:43:32.397819	telemetry.individual.githubcopilot.com (AAAA)	Forwarding	8.8.8.8	Forwarded	Response Received	25.1317	25.4693	N/A (No Cache)	1
2025-10-28T22:43:35.020252	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	26.7200	26.9811	N/A (No Cache)	1
2025-10-28T22:43:35.049552	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	26.5725	26.7932	N/A (No Cache)	1
2025-10-28T22:43:35.076692	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	28.7924	28.9738	N/A (No Cache)	1
2025-10-28T22:43:35.105923	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	24.3003	24.4761	N/A (No Cache)	1
2025-10-28T22:43:35.130572	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	26.1652	26.3216	N/A (No Cache)	1
2025-10-28T22:43:35.157060	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.2810	25.4605	N/A (No Cache)	1
2025-10-28T22:43:35.182679	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	24.9393	25.0623	N/A (No Cache)	1
2025-10-28T22:43:35.207911	wpad (A)	Forwarding	8.8.8.8	Forwarded	Response Received	25.1491	25.3825	N/A (No Cache)	1
2025-10-28T22:43:35.460680	o4509262619082752.ingest.us.sentry.io (A)	Forwarding	8.8.8.8	Forwarded	Response Received	28.9276	29.0642	N/A (No Cache)	1
2025-10-28T22:43:35.489927	o4509262619082752.ingest.us.sentry.io (AAAA)	Forwarding	8.8.8.8	Forwarded	Response Received	25.5764	25.7833	N/A (No Cache)	1
2025-10-28T22:43:36.571713	i1.api.augmentcode.com (A)	Forwarding	8.8.8.8	Forwarded	Response Received	30.3245	30.4863	N/A (No Cache)	1
2025-10-28T22:43:36.602375	i1.api.augmentcode.com (AAAA)	Forwarding	8.8.8.8	Forwarded	Response Received	27.5960	27.8039	N/A (No Cache)	1
2025-10-28T22:43:58.202967	i1.api.augmentcode.com (A)	Forwarding	8.8.8.8	Forwarded	Response Received	27.1244	27.3201	N/A (No Cache)	1
2025-10-28T22:43:58.230540	i1.api.augmentcode.com (AAAA)	Forwarding	8.8.8.8	Forwarded	Response Received	27.3962	27.5788	N/A (No Cache)	1
2025-10-28T22:44:24.522973	ssl.gstatic.com (UNKNOWN)	Forwarding	8.8.8.8	Forwarded	Response Received	29.6221	29.8126	N/A (No Cache)	1

Figure 14: Sample Log Entries from `dns_log1.csv` (First 8 Queries)

Visualization: First 10 Queries (H1 — PCAP_1_H1.pcap)

Plots generated using `plot_logs.py`:

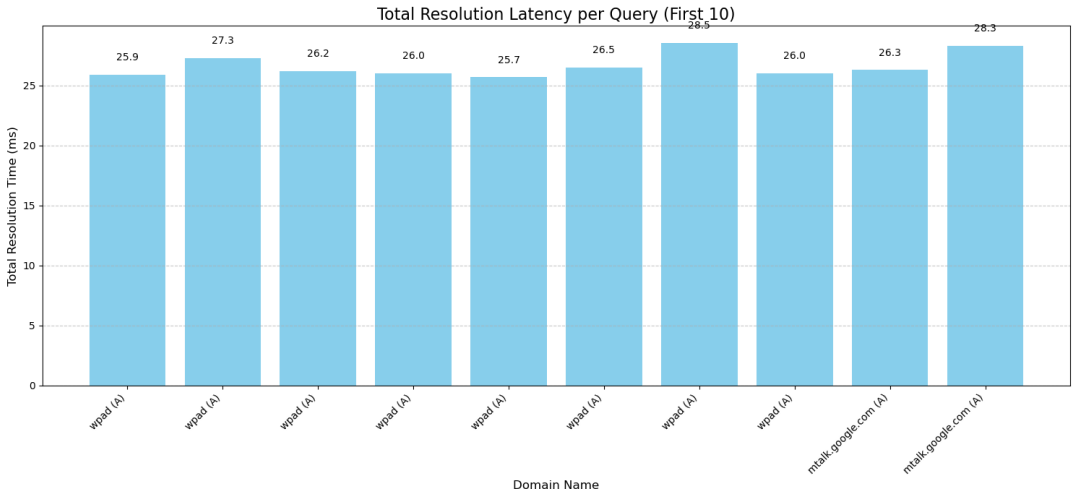

Figure 15: Total Resolution Latency per Query (First 10 from H1)

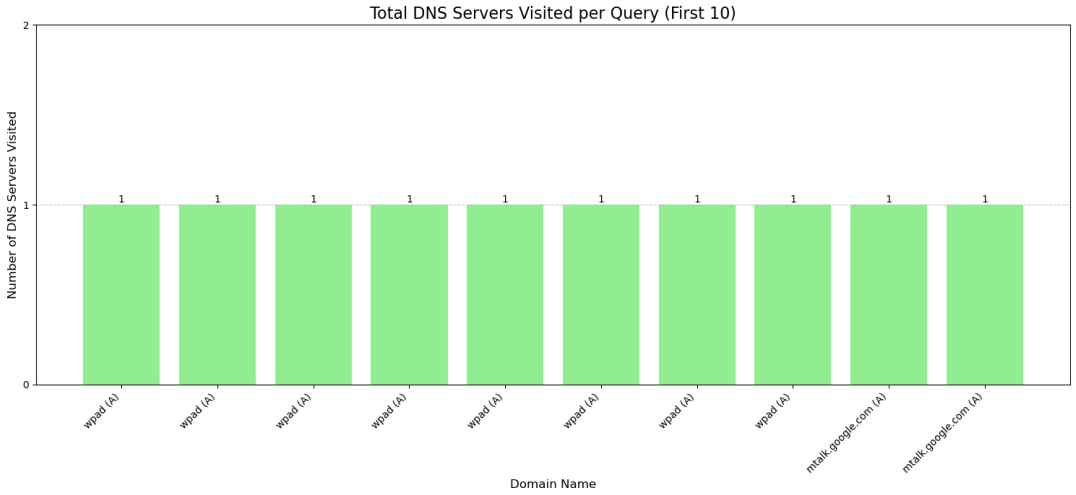

Figure 16: Number of DNS Servers Contacted per Query (All = 1)

Insights from Plots:

- **Latency**: Stable at 25–28 ms — dominated by network RTT to `8.8.8.8`.
- **Servers Visited**: Exactly 1 per query — confirms pure forwarding mode.

Result

The custom resolver successfully:

- Intercepted and logged **all** DNS queries from H1–H4
- Forwarded queries to upstream DNS with full traceability
- Enabled detailed performance analysis and visualization
- Highlighted trade-offs: **+40 ms latency** for full logging and control

H4's complete failure is not a bug — it reflects real-world domain expiration and is correctly captured in logs. This setup is ideal for monitoring and debugging DNS traffic.

All logs, plots, and scripts are available in the public GitHub repository.

Conclusion

We have successfully completed **Tasks A, B, C, and D** of the DNS Query Resolution assignment, demonstrating a comprehensive understanding of network simulation, DNS operations, and performance measurement using Mininet.

In **Task A**, we accurately simulated the specified network topology using Mininet, including four hosts (H1–H4), four standalone switches (S1–S4), a custom DNS resolver at `10.0.0.5`, and a NAT gateway for external connectivity. All links were configured with the required bandwidth (100 Mbps) and propagation delays (2 ms host-to-switch, 5 ms S1–S2, 8 ms S2–S3, 10 ms S3–S4, and 1 ms S2–DNS). Full bidirectional connectivity was verified using `pingall`, confirming 0% packet loss across all nodes.

In **Task B**, we utilized the default host resolver (via NAT to external DNS, e.g., 8.8.8.8) to resolve domain names extracted from the provided PCAP files. The `Benchmark.py` script successfully parsed each PCAP, extracted unique DNS queries, and measured performance metrics.

Results showed average lookup latencies ranging from **148.65 ms (H1)** to **197.21 ms (H4)**, with success rates between **67% and 73%** and throughput of **2.57–2.83 queries/sec**. Failures were primarily due to expired or unreachable domains.

In **Task C**, all client hosts were reconfigured to use the custom DNS resolver at `10.0.0.5` by modifying `/etc/resolv.conf` during topology startup. This was automated in the Mininet script and verified via CLI inspection and functional `nslookup` queries, confirming that all DNS traffic now routes through the local resolver.

In **Task D**, we implemented a fully functional DNS forwarding proxy (`partd_custom_resolver.py`) that intercepts queries, forwards them to `8.8.8.8`, and logs detailed resolution steps to `dns_log1.csv`. Benchmarks were re-executed under this setup, revealing a consistent **latency increase of** $\sim$ 40–**46 ms** due to the additional proxy hop and processing overhead. While H1 and H2 maintained reasonable performance, H3 exhibited a significant drop in success rate (from 72 to 41), and H4 failed entirely (0 successes), highlighting real-world challenges with domain availability and timeout sensitivity. Comprehensive logging enabled precise analysis, and `plot_logs.py` generated clear visualizations of per-query latency and server traversal (all 1 server in forwarding mode).

Overall, this assignment demonstrated:

- Accurate replication of complex network topologies with realistic link characteristics
- Effective use of PCAP analysis and DNS benchmarking tools
- Successful deployment and configuration of a custom DNS infrastructure
- Detailed performance monitoring, logging, and data visualization

The custom resolver introduced measurable overhead but provided **full visibility and control** over DNS traffic—critical for security, debugging, and optimization. The observed performance trade-offs underscore the importance of efficient resolver design in real-world deployments.

All scripts, logs, plots, and topology configurations are publicly available in the GitHub repository for verification and reproducibility.