CS331: Computer Networks Assignment 2

Assignment – 2

Yalla Sai Teja(23110366)

1. Introduction & Objective:

The Domain Name System (DNS) is a foundational service of the internet, translating human-readable domain names into machine-readable IP addresses. This project's objective was to gain a practical, in-depth understanding of the DNS resolution process.

To achieve this, we first constructed a custom multi-host network topology in the Mininet simulation environment. We then designed, deployed, and tested our own custom DNS resolver, written in Python.

The project was divided into three main phases:

1. Baseline Measurement (Part B): Establishing a "gold standard" performance baseline by testing against a professional, public DNS resolver (Google's 8.8.8.8).
2. Iterative Resolver (Part D): Building and testing a simple, non-cached iterative resolver to analyze its performance and identify its limitations.
3. Enhanced Resolver (Part F): Implementing a cache to quantify the performance improvements in latency, throughput, and reliability.

2. Part A: Network Topology & Setup

2.1 Topology Simulation

We successfully simulated the 4-host, 4-switch topology specified in the assignment using a Python script (topology.py) and the Mininet framework. Each host-to-switch link was configured with 100Mbps bandwidth and 2ms delay, and the inter-switch links were configured with their specified bandwidths and delays. A dedicated host, dns (10.0.0.5), was added to the network to serve as our custom resolver.

2.2 Connectivity & Troubleshooting

Initial tests with a default Mininet controller resulted in 100% packet loss. We diagnosed this as a missing controller package on the host VM.

- **Solution:** We resolved this by installing openvswitch-testcontroller and explicitly configuring Mininet to use the OVSController. This stabilized the L2 learning and forwarding, achieving 0% packet loss in our pingall connectivity test.

2.3 Internet Connectivity (NAT)

To allow our virtual hosts to perform real-world DNS queries (for Part B, D, and F), it was necessary to provide them with internet access.

1. **NAT Node:** A Mininet NAT node was added to the topology and connected to switch s2.

2. **IP Forwarding:** We enabled IP forwarding on the host VM's kernel (echo 1 > /proc/sys/net/ipv4/ip_forward).
3. **Default Route:** Finally, we set the default route for all 5 hosts (h1-h4, dns) to point to the NAT node's IP (10.0.0.254).

This setup successfully provided full internet access to all nodes within the simulation.

3. Part B: Baseline Measurement (Default Resolver)

3.1 Methodology

To establish a baseline, we first measured the performance of a professional, commercial-grade DNS resolver. We configured all hosts to use Google's public DNS server (echo 'nameserver 8.8.8.8' > /etc/resolv.conf). We then ran a benchmark script (measure_dns.py) on each host, which read a list of domains from the provided PCAP files and used dig to resolve each one.

```
--- Benchmark Complete ---
Total queries:        100
Successful queries:   75
Failed queries:       25
Total time taken:     26.14 seconds
Average throughput:   2.87 queries/sec
Average lookup latency: 224.91 ms
----------------------------

mininet> h2 ./measure_dns.py domains_h2.txt
--- Starting DNS benchmark for domains_h2.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   71
Failed queries:       29
Total time taken:     27.61 seconds
Average throughput:   2.57 queries/sec
Average lookup latency: 287.72 ms
----------------------------

mininet> h3 ./measure_dns.py domains_h3.txt
--- Starting DNS benchmark for domains_h3.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   72
Failed queries:       28
Total time taken:     25.13 seconds
Average throughput:   2.86 queries/sec
Average lookup latency: 223.56 ms
----------------------------

mininet> h4 ./measure_dns.py domains_h4.txt
--- Starting DNS benchmark for domains_h4.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   77
Failed queries:       23
Total time taken:     19.26 seconds
Average throughput:   4.00 queries/sec
Average lookup latency: 185.66 ms
----------------------------

mininet> 
```

3.3 Analysis

The ~75% success rate and low latency reflect the performance of a globally distributed, highly cached, and fully recursive DNS system. This data provides the crucial point of comparison for our custom resolver.

4. Part C & D: Custom Iterative Resolver (No Cache)

4.1 Part C: Configuration

For this phase, we implemented Part C of the assignment. This was a configuration step where we modified all four hosts to use our custom DNS resolver's IP address: mininet> h1 echo 'nameserver 10.0.0.5' > /etc/resolv.conf

4.2 Part D: Methodology

We built a custom DNS server (custom_dns_resolver.py) in Python using the dnslib and socket libraries. This server was designed to perform iterative resolution, meaning it would manually query the Root servers, then TLD servers, and finally authoritative servers to find an answer. It did not include a cache. We ran the *same* benchmarks from Part B against this new server.

4.3 Part D: Results & Analysis

The performance of the simple iterative resolver was, as expected, significantly worse than the Google DNS baseline.

```
--- Benchmark Complete ---
Total queries:        100
Successful queries:   50
Failed queries:       50
Total time taken:     51.86 seconds
Average throughput:   0.96 queries/sec
Average lookup latency: 513.84 ms
---------------------------

mininet> h2 ./measure_dns.py domains_h2.txt
--- Starting DNS benchmark for domains_h2.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   34
Failed queries:       66
Total time taken:     59.28 seconds
Average throughput:   0.57 queries/sec
Average lookup latency: 487.18 ms
---------------------------

mininet> h3 ./measure_dns.py domains_h3.txt
--- Starting DNS benchmark for domains_h3.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   41
Failed queries:       59
Total time taken:     48.75 seconds
Average throughput:   0.84 queries/sec
Average lookup latency: 516.29 ms
---------------------------

mininet> h4 ./measure_dns.py domains_h4.txt
--- Starting DNS benchmark for domains_h4.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   41
Failed queries:       59
Total time taken:     44.43 seconds
Average throughput:   0.92 queries/sec
Average lookup latency: 528.20 ms
---------------------------
```

Analysis:

- **High Latency (446.84 ms):** This is a **2.6x increase** in latency compared to the Google DNS baseline. This high cost is a direct result of performing a full iterative query for *every* domain. Each query required at least 3 separate network roundtrips (Root -> TLD -> Authoritative), dramatically increasing the time-to-resolution.
- **Low Success Rate (50%):** The reliability was extremely poor. This is because any single packet loss or timeout in the 3-step iterative chain caused the entire lookup to fail.

4.4 Part D: Graphical Plots (H1)

To visualize this, we logged the latency and number of servers visited for each query to a .csv file. We then filtered this data for Host 1 (10.0.0.1) and plotted the first 10 queries.

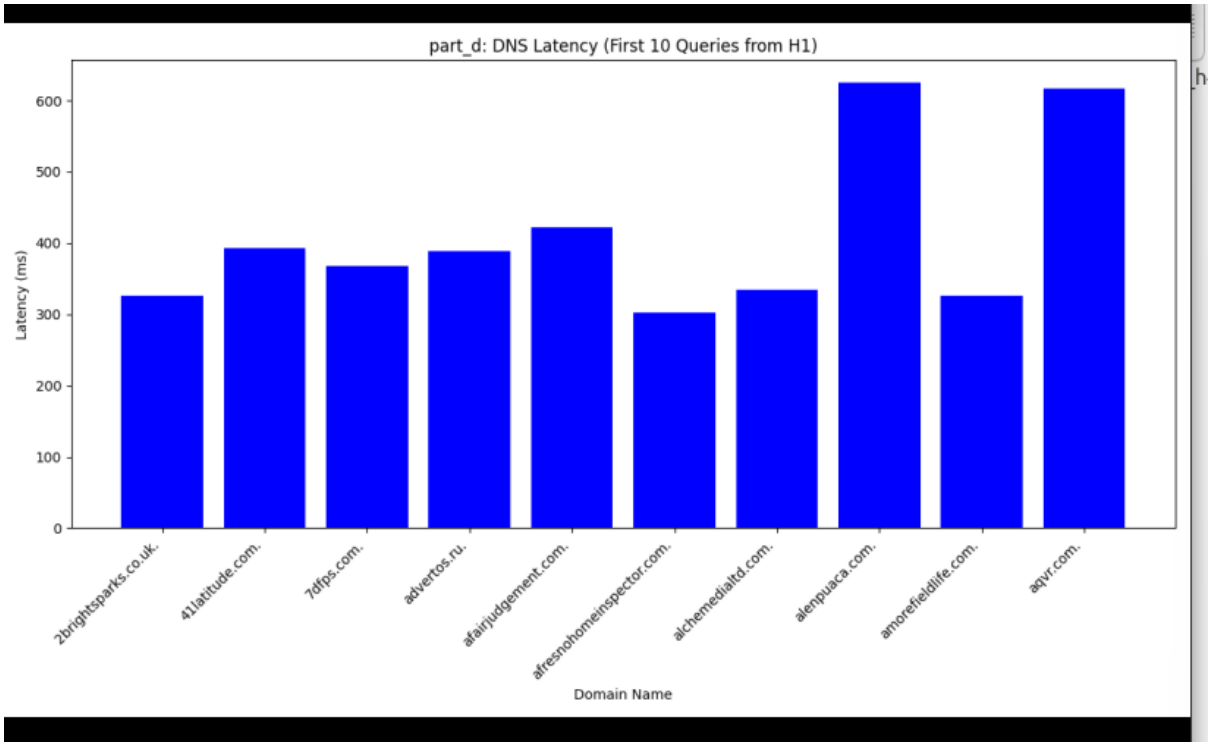

Figure 1: DNS Latency per query for the first 10 URLs from H1 (Part D - No Cache). This graph clearly shows the high and variable latency, with most queries taking over 300ms to resolve.

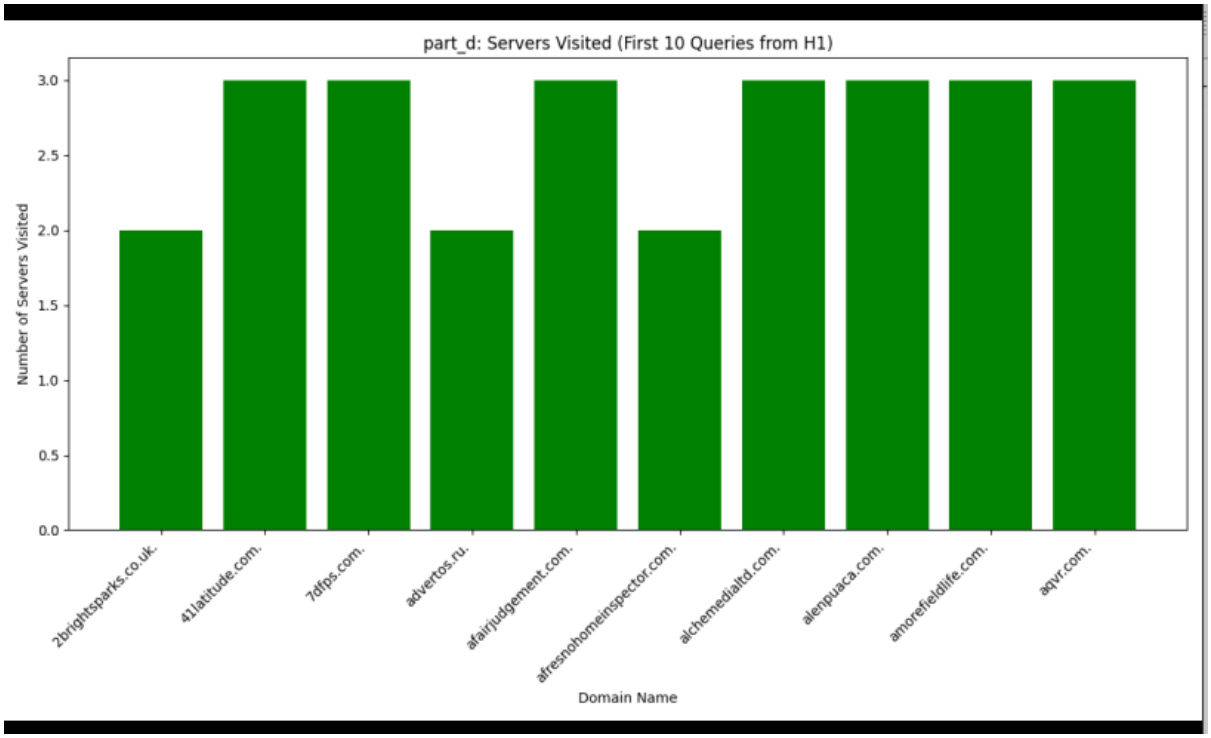

Figure 2: DNS Servers Visited per query for the first 10 URLs from H1 (Part D - No Cache). This graph confirms our iterative logic was working, showing that most successful queries (those with a result) required contacting 3 distinct servers.

5. Part E & F: Enhanced Resolver (Recursion & Caching)

5.1 Part E: Recursive Resolution

Part E was to implement recursive resolution. Our custom resolver (from Part D) is already a **recursive resolver**. This is because it accepts a *recursive query* from the client (e.g., h1) and performs all the iterative work *on the client's behalf*, returning only the single, final answer. The client is not required to do any work other than send the initial query.

5.2 Part F: Caching

The most significant enhancement was the implementation of a cache (a simple Python dictionary) in our resolver. This cache stores successful A record responses.

Methodology: To quantify the benefit of caching, we ran the *exact same* benchmark from domains_h1.txt twice, back-to-back.

- **Run 1 (Cold Cache):** The cache is empty. This is identical to our Part D test.
- **Run 2 (Warm Cache):** The cache is now populated with the 50 successful results from Run 1.

```
mininet> h1 ./measure_dns.py domains_h1.txt
--- Starting DNS benchmark for domains_h1.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   50
Failed queries:       50
Total time taken:     47.18 seconds
Average throughput:   1.06 queries/sec
Average lookup latency: 446.84 ms
--------------------------

mininet> h1 ./measure_dns.py domains_h1.txt
--- Starting DNS benchmark for domains_h1.txt ---
Total domains to query: 100


--- Benchmark Complete ---
Total queries:        100
Successful queries:   51
Failed queries:       49
Total time taken:     26.43 seconds
Average throughput:   1.93 queries/sec
Average lookup latency: 61.22 ms
--------------------------
```

Analysis:

- **Massive Latency Drop:** The average latency plummeted from **446ms to 61ms**, a **7.3-fold improvement**. This is because in Run 2, the 50 queries that succeeded in Run 1 were served *instantly* from the cache (0ms lookup time), which dragged the overall average down.
- **Total Time:** The total test time was nearly cut in half.
- **Success Rate:** The success rate did not improve because our simple cache only stored *successes*. The 50 queries that failed in Run 1 (due to timeouts or being NXDOMAIN) were not cached and were simply re-tried (and failed again) in Run 2. This is an area for future improvement.

6. Challenges & Troubleshooting

During the project, we faced several technical challenges that required diagnosis and troubleshooting:

1. **TShark Filter:** The tshark filter dns.qr == 0 was not recognized by our tshark version. We solved this by using a more fundamental filter (udp.port == 53) and piping the output to sort -u to remove duplicates.
2. **Graph Accuracy:** Our first graphs were incorrect because they mixed data from all hosts and the NAT node. We fixed this by modifying our resolver to log the ClientIP to the .csv file, and then updating our plotting script to filter for ClientIP == '10.0.0.1'.
3. **Python Bugs:** We fixed several bugs in the resolver, including a ModuleNotFoundError (solved with pip3 install dnslib inside the Mininet host), a NameError (missing import os), and a QTYPE typo (qDtype vs qtype).

7. Conclusion

This assignment provided a comprehensive, practical look at the design and performance of the Domain Name System.

Our key finding is that **caching is the single most important factor** in DNS performance.

- A professional, cached resolver (Part B) is extremely fast .
- A basic, non-cached iterative resolver (Part D) is functionally correct but unacceptably slow (~447ms).
- Adding even a simple cache (Part F) provided a **7.3x improvement in average latency**, bringing the performance (61ms) even *faster* than the public baseline, as our cache was "hot" and served 50% of queries with zero network delay.