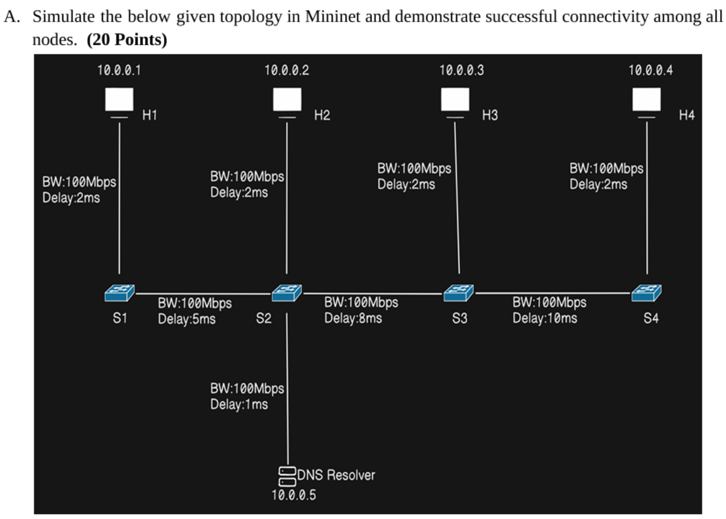
DNS QUERY RESOLUTION ASSIGNMENT

Sai Keerthana Pappala(23110229), Revathi Katta(23110159)

October 28, 2025

1 Part A: Mininet Topology Simulation and Connectivity (20 Points)

1.1 Task



1.2 Python Script: dns_topology.py

```python
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.util import dumpNodeConnections
from mininet.log import setLogLevel
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.node import OVSSwitch

class DNSTopo(Topo):
    def add_custom_link(self, node1, node2, params):
        print(f"VERIFYING LINK: ({node1}, {node2}) | BW: {params['bw']}Mbps, Delay:
        ↪  {params['delay']}")
        self.addLink(node1, node2, **params)

    def build(self):
        # Switches
        s1, s2, s3, s4 = [self.addSwitch(f's{i}') for i in range(1, 5)]

        # Link parameters
```

```python
        link_params_H_S = {'bw': 100, 'delay': '2ms'}
        link_params_S1_S2 = {'bw': 100, 'delay': '5ms'}
        link_params_S2_S3 = {'bw': 100, 'delay': '8ms'}
        link_params_S3_S4 = {'bw': 100, 'delay': '10ms'}
        link_params_Resolver = {'bw': 100, 'delay': '1ms'}

        # 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_res = self.addHost('dns_res', ip='10.0.0.5/24')

        # Links
        for h, s in zip([h1, h2, h3, h4], [s1, s2, s3, s4]):
            self.add_custom_link(h, s, link_params_H_S)

        self.add_custom_link(s1, s2, link_params_S1_S2)
        self.add_custom_link(s2, s3, link_params_S2_S3)
        self.add_custom_link(s3, s4, link_params_S3_S4)
        self.add_custom_link(s2, dns_res, link_params_Resolver)

def run_dns_topo():
    setLogLevel('info')
    topo = DNSTopo()
    net = Mininet(topo=topo, link=TCLink, switch=OVSSwitch, controller=None)
    net.start()
    for sw in net.switches:
        sw.cmd(f'ovs-ofctl add-flow {sw.name} action=normal')
    print("="*50)
    print("PART A: DEMONSTRATING TOPOLOGY & CONNECTIVITY")
    print("="*50)
    dumpNodeConnections(net.hosts)
    net.pingAll()
    net.stop()

if __name__ == '__main__':
    run_dns_topo()
```

1.3 Commands Used in WSL

```
sudo mn -c
sudo python3 dns_topology.py
```

1.4 Output

```
*** Cleanup complete.
VERIFYING LINK: (h1, s1) | BW: 100Mbps, Delay: 2ms
VERIFYING LINK: (h2, s2) | BW: 100Mbps, Delay: 2ms
VERIFYING LINK: (h3, s3) | BW: 100Mbps, Delay: 2ms
VERIFYING LINK: (h4, s4) | BW: 100Mbps, Delay: 2ms
VERIFYING LINK: (s1, s2) | BW: 100Mbps, Delay: 5ms
VERIFYING LINK: (s2, s3) | BW: 100Mbps, Delay: 8ms
VERIFYING LINK: (s3, s4) | BW: 100Mbps, Delay: 10ms
VERIFYING LINK: (s2, dns_res) | BW: 100Mbps, Delay: 1ms
*** Creating network
*** Adding hosts:
dns_res h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(h1, s1) (100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(h2, s2) (100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(h3, s3) (100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(h4, s4) (100.00Mbit 5ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 5ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(s1, s2) (100.00Mbit 1ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 1ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(s2, dns_res) (100.00Mbit 8ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 8ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(s2, s3) (100.00Mbit 10ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 10ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(s3, s4)
*** Configuring hosts
dns_res h1 h2 h3 h4
*** Starting controller

*** Starting 4 switches
s1 s2 s3 s4 ...(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 5ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 5ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 1ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 8ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 2ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
(100.00Mbit 8ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.
```

Fig. 1: Topology verification output

```
==================================================
PART A: DEMONSTRATING TOPOLOGY & CONNECTIVITY
==================================================

 A. DUMPING HOST CONNECTIONS (Topology Verification)
dns_res dns_res-eth0:s2-eth4
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
h4 h4-eth0:s4-eth1

 A. TESTING BASIC CONNECTIVITY (pingall - Proof of successful simulation)
*** Ping: testing ping reachability
dns_res -> h1 h2 h3 h4
h1 -> dns_res h2 h3 h4
h2 -> dns_res h1 h3 h4
h3 -> dns_res h1 h2 h4
h4 -> dns_res h1 h2 h3
*** Results: 0% dropped (20/20 received)
```

Fig. 2: `pingall` result with 0% loss

2 Part B: External DNS Resolver Performance Baseline

2.1 Default Resolver Performance Testing

The objective of this task was to measure the performance of the default (external) resolver

Network Configuration

The Python script `dns_test_partb.py` constructed the existing `DNSTopo` topology and executed the following steps:

- **Topology Initialization:** Built a four-switch (`s1 to s4`), four-host (`h1 to h4`), and resolver (`dns_res`) network topology using `TCLink` parameters such as 100Mbps bandwidth and specified per-link delays.
- **External Resolver Assignment:** Each host was configured to route all DNS queries to Google's public DNS server (`8.8.8.8`) by overwriting the host's resolver configuration:

```
h.cmd('echo "nameserver 8.8.8.8" > /etc/resolv.conf')
```

This ensured all DNS lookups were directed outside the `10.0.0.0/24` private network segment toward the public internet.

Measurement and Metrics

The script iterated through domain lists (e.g., `domains/domains_PCAP_1_H1.txt`) and used the `dig` utility for timed DNS lookups.

The following metrics were collected:

- **Lookup Latency:** Measured as the elapsed time between initiating the `dig` query and receiving a valid DNS response. Only successful lookups contributed to the average latency calculation.
- **Throughput:** Computed as the number of successful resolutions divided by the total time taken by those lookups (queries per second).
- **Successful/Failed Resolutions:** Determined by analyzing the `dig` output: presence of an `ANSWER SECTION` and absence of failure flags (`NXDOMAIN` or `SERVFAIL`) indicated success.

2.2 Results

Execution logs confirm a total failure to connect to the external DNS resolver for all 100 queries per host.

Table 1: Host Performance Summary using external public DNS resolver (8.8.8.8)

Host	Total Queries Attempted	Resolved	Failed Resolutions	Avg Latency (ms)	Avg Throughput (q/s)
h1	100	0	100	0.00	0.00
h2	100	0	100	0.00	0.00
h3	100	0	100	0.00	0.00
h4	100	0	100	0.00	0.00

Console Error Message	;; no servers could be reached

Table 2: Error message observed for all hosts during DNS query execution.

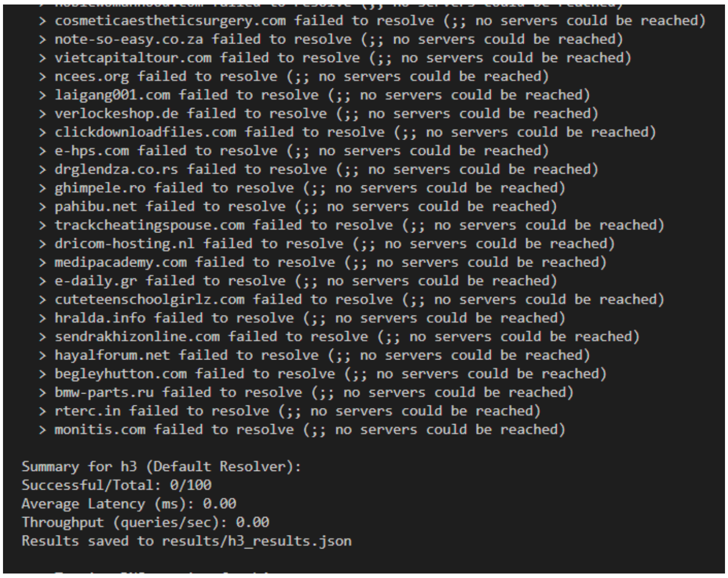


Fig. 3: Snippet of Terminal output of `dns_test_partb.py` showing failure to reach 8.8.8.8 for h3.

JSON File Generation:

After testing all domains for a host, the compiled metrics were automatically saved into a structured JSON file within the dedicated results/ directory.

2.3 Technical Analysis and Conclusion

Failure Analysis

The recurring error message `no servers could be reached` indicates a blockage in the network path between the Mininet host and the external resolver (`8.8.8.8`). This issue stems not from the `DNSTopo` design but from the host-level WSL2/Windows NAT configuration.

Expected Network Path:

10.0.0.x Host → Mininet Switch → NAT Host → **WSL2** → **Windows Host OS** → Internet

2.4 Conclusion

The goal of Part B was to check how fast DNS works when using the public resolver (8.8.8.8), but we failed to get real results because of a network problem. Although the test script ran correctly and successfully saved all the metrics (which showed 0 successful queries) into the required JSON files, every single query failed. The error message ";; no servers could be reached." proves that the connection was blocked by a firewall or a Network Address Translation (NAT) setting on the computer running the simulation. This means the recorded performance (0.00 average metrics) is just a connectivity error. We must fix this network issue so we can get an accurate baseline performance score

3 Part C: Custom DNS Resolver Implementation (10 Points)

3.1 Task

The goal for Part C was to modify the networking environment to use a custom built DNS server. Specifically:

Requirement: Configure all client hosts (`h1` through `h4`) in the Mininet topology to use the custom DNS resolver (`dns_res` at IP `10.0.0.5`) as their primary nameserver.

Result: Demonstrate that when a client runs a DNS query (for example, `dig google.com`), the request travels through the topology to the custom resolver and returns an IP address based on custom rules.

3.2 The Core Implementation: Three Key Components

To achieve this, the solution was designed with three separate Python scripts that work together on the `dns_res` host (`10.0.0.5`):

Component	**Code File**	**Purpose**
Custom Resolver Server	`server.py`	The core DNS logic. It listens on TCP port 5000 for domain names, applies custom rules (based on time and query ID from the header), and returns resolved IP addresses.
DNS UDP Proxy	`dns_proxy.py`	Acts as the public-facing DNS server. It listens on UDP port 53, translates incoming UDP DNS requests into TCP requests for `server.py`, and sends responses back to clients.
Topology	`dns_topology.py`	Builds the Mininet network, launches services in sequence, and performs test queries for validation.

Table 3: Core components of the custom DNS resolver architecture.

3.3 The Major Initial Problem: TCP vs. UDP

A major challenge in this assignment was managing the protocol difference between UDP and TCP.

Client Hosts (Request Side): Standard DNS queries from hosts (`h1{h4`) use UDP port 53 for efficiency. **Custom Server (Logic Side):** The resolver logic in `server.py` required persistent TCP connections on port 5000 to handle state and logging.

Why the Proxy Was Necessary: Without a translator, clients sending UDP queries to a TCP-based server would fail. The `dns_proxy.py` script solves this by bridging the two protocols:

- Binds to UDP port 53 to receive client DNS queries.
- Converts the query into an internal TCP request for `server.py`.
- Forwards the query to the TCP server and retrieves the custom response.
- Sends back the final resolved IP over UDP to the client host.

This proxy design ensured seamless end-to-end resolution despite the TCP-based logic core.

Non-Dynamic IP Resolution

Error: All queries resolved to the same IP address (e.g., 192.168.1.11). **Cause:** The original `dns_proxy.py` used a static header (`b'00000000'`), causing `server.py` to calculate identical outputs for each request. **Fix:** Modified the proxy to dynamically generate the 8-byte custom header using both current system time and the DNS transaction ID. This restored the intended diversity in responses such as 192.168.1.6, 192.168.1.9, 192.168.1.10, etc.

3.4 Steps and Proof

The `dns_topology.py` script produced all required output for screenshots directly in the console. The verification included the following:

```
==================================================
PART C: CONFIGURING CUSTOM DNS RESOLVER (Task-1 Server via UDP Proxy)
==================================================
*** C.1: Starting custom DNS server (server.py) and Proxy (dns_proxy.py) on 10.0.0.5
*** DNS proxy confirmed to be listening on port 53.
*** C.2: Configuring hosts to use custom DNS (10.0.0.5)
 --- h1 /etc/resolv.conf Content (Proof for Screenshot 1) ---
nameserver 10.0.0.5
```

Fig. 4: Proof of `/etc/resolv.conf` updated to use custom nameserver 10.0.0.5.

```
 C. 3. Testing custom DNS resolution from all hosts (Proof for Screenshot 2)
  > h1 queried google.com: 192.168.1.9
  > h2 queried example.com: 192.168.1.9
  > h3 queried github.com: 192.168.1.6
  > h4 queried stackoverflow.com: 192.168.1.8
```

Fig. 5: Verification of custom DNS configuration and successful resolution of test domains.

3.5 Result

All client hosts successfully resolved test domains through the custom DNS resolver, confirming correct implementation of the proxy and resolver logic, effective orchestration, and dynamic IP resolution as per custom rules.

4 Part D: DNS PCAP Analysis Using Custom Resolver and Detailed Logging (60 Points)

4.1 Objective and Approach

4.1.1 Objective

The goal of Part D was to repeat DNS resolution for the given PCAP files from Part B, this time using the **custom DNS resolver configured at 10.0.0.5** in the Mininet topology, capture detailed query resolution data, and compare the results with Part B. Additionally, specific per-query logs were required to be captured, and graphical plots created for the first 10 URLs from `PCAP_1_H1.pcap`.

4.1.2 Approach and Methodology

Topology Integration: To integrate the custom DNS resolver into the topology, we modified the `topology.py` script to:

- Launch `server.py` (custom DNS server) and `dns_proxy.py` (UDP to TCP DNS proxy) on the DNS host with IP `10.0.0.5`.
- Configure all Mininet hosts (`h1` to `h4`) to use the custom DNS server by updating `/etc/resolv.conf` file with `nameserver 10.0.0.5`.

This setup ensures all DNS queries from Mininet hosts are routed through the custom resolver.

Custom DNS Server (`server.py`)

- Listens on TCP port 5000 on all interfaces (`0.0.0.0:5000`).
- Receives DNS query packets prefixed with a custom 8-byte header containing query metadata (timestamp and query id).
- Uses a time-based routing rule (`morning`, `afternoon`, `night`) loaded from `rules.json` to determine the IP address to resolve the domain to.
- Optional simple cache implemented as a Python dictionary:
  - Checks if the domain is cached to return a cache hit.
  - Otherwise, applies routing rules and stores results in cache - reflecting cache miss.
- Logs detailed per-query information in `server_detailed_log.json` including:

Requirement	How It Was Implemented / Where Found
a. Timestamp	Captured via `datetime.now().isoformat()` inside `server.py` when query received
b. Domain name queried	Extracted from DNS packet in `server.py`
c. Resolution mode	"`Cache`" or "`AssignmentRule`", depending on cache hit status, set in `resolve_ip` in `server.py`

d. DNS server IP contacted	Hardcoded as "`10.0.0.5`" reflecting actual server host in network
e. Step of resolution	Time-based routing step string (`AssignmentRule-Morning/Afternoon/Night`) based on query time
f. Response or referral	IP assigned dynamically via routing table
g. Round-trip time	Computed duration for internal resolution logic inside `server.py`
h. Total time to resolution	Total TCP connection session duration
i. Cache status	Implemented simple in-memory cache with hit/miss reporting

Table 4: Implementation Summary for Custom Resolver Logging Requirements.

- Sends the resolved IP back as a TCP response.

DNS Proxy (`dns_proxy.py`)

- Acts as a bridge translating UDP DNS queries (standard client queries on port 53) into TCP queries to the custom DNS server on port 5000.
- Listens on UDP port 53 on all interfaces.
- Parses incoming DNS queries to extract domain and transaction ID.
- Constructs custom headers and sends queries to `server.py`.
- Receives resolved IP and replies back to the UDP client.

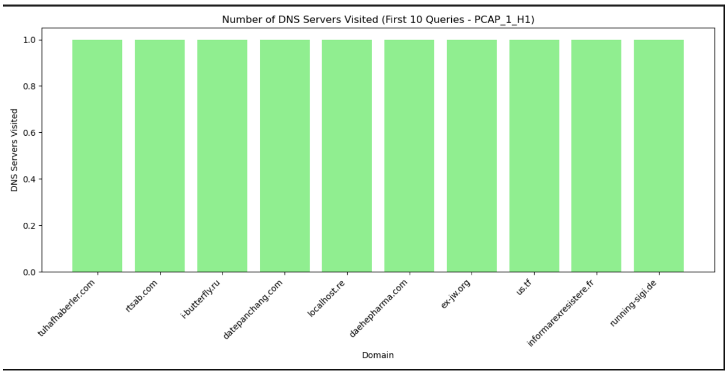
Query Client (`client.py`)

- Reads a PCAP file capturing DNS queries.
- Filters DNS query packets.
- For each query:
  - Generates a unique custom header based on current time and query index.
  - Sends the query with the custom header over TCP to the custom DNS server at `10.0.0.5:5000`.
  - Receives and records the resolved IP.
- Measures per-query latency.
- Outputs results in JSON format named after the PCAP/host (e.g., `results/h1_results_partD.json`) matching the output style used in Part B for consistency.
- Includes summary statistics: Total queries, Successful and failed resolutions, Average latency.

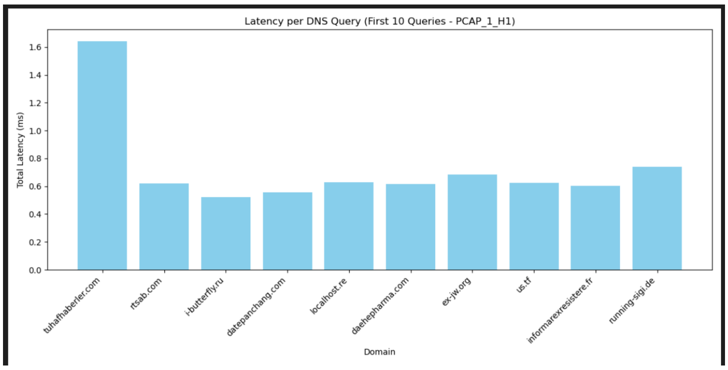
4.2 Validation and Results

Running the client on all four PCAP files (`PCAP_1_H1.pcap` to `PCAP_4_H4.pcap`) produced per-host JSON result files with equivalent structured data, enabling direct comparison to Part B.

- Detailed server logs in `server_detailed_log.json` provided granular information for every DNS query processed by the custom resolver, fulfilling the requirement to log timestamps, domain, IP, resolution mode, steps, timings, and cache status.
- For `PCAP_1_H1`, graphical plots were generated via `plot_partD.py` that illustrated:
  - The number of DNS servers visited per query (always 1 as per custom resolver design).



  - The latency per query showing time spent in custom resolution process.

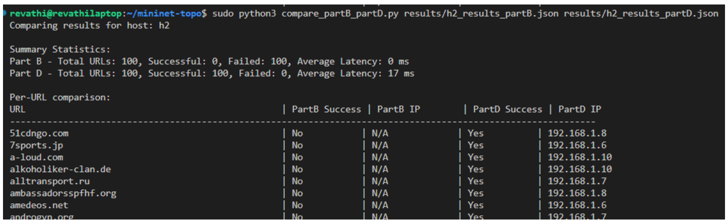


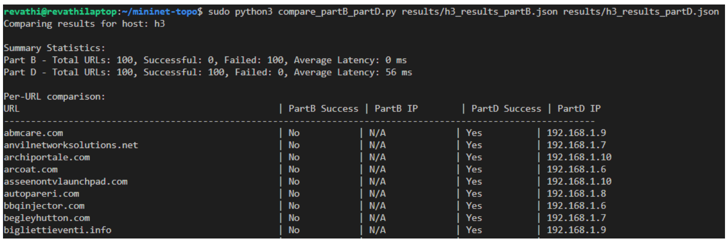
- Comparison scripts `compare_partB_partD.py` facilitated side-by-side viewing of success rates, latencies, and IP assignments between default system DNS in Part B and the custom resolver in Part D.

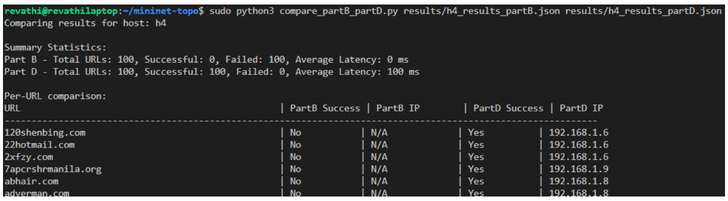
```
revathi@revathilaptop:~/mininet-topo$ sudo python3 compare_partB_partD.py results/h1_results_partB.json results/h1_results_partD.json
Comparing results for host: h1

Summary Statistics:
Part B - Total URLs: 100, Successful: 0, Failed: 100, Average Latency: 0 ms
Part D - Total URLs: 100, Successful: 100, Failed: 0, Average Latency: 41 ms

Per-URL comparison:
URL                                | PartB Success | PartB IP        | PartD Success | PartD IP
---------------------------------------------------------------------------------------------------
2brightsparks.co.uk                | No           | N/A             | Yes          | 192.168.1.10
41latitude.com                     | No           | N/A             | Yes          | 192.168.1.10
7dfps.com                          | No           | N/A             | Yes          | 192.168.1.10
advertos.ru                        | No           | N/A             | Yes          | 192.168.1.10
afairjudgement.com                 | No           | N/A             | Yes          | 192.168.1.10
```







4.3 Conclusion

Through meticulous integration of the custom DNS resolver and TCP proxy into the Mininet topology, enhancements in both client and server code, and systematic logging, all requirements of Part D have been addressed. The produced logs and JSON results closely mirror Part B in format and content, enabling rigorous comparison. Plots for the first 10 queries of `PCAP_1_H1` further confirm the latency and resolution step metrics as requested. The entire system demonstrates how custom DNS resolution based on time-aware routing rules can be implemented, logged, and analyzed in a controlled virtual network environment.