Computer Networks Assignment 2

Gaurav Budhwani (22110085) Kaveri Visavadiya (22110114)

Tuesday 28th October, 2025

Instructor: Sameer Kulkarni

Contents

1 Task A — Network Topology Simulation using Mininet

1.1 Objective

The objective of this task is to simulate the topology given in the assignment diagram on Mininet using a Virtual Machine. The goal is to demonstrate successful connectivity all the nodes (hosts `h1-h4`, switches `s1-s4`, `dns` resolver).

1.2 Mininet Installation on macOS

We used Multipass to create an Ubuntu VM on macOS (to provide an isolated Linux environment to run Mininet).

1. **Install Multipass using Homebrew:**

```
brew update
brew install --cask multipass
```

2. **Launch and configure Ubuntu VM:** Our 'mininet' instance is allocated 2 CPUs, 4GB RAM & 5GB diskspace.

```
multipass launch --name mininet --cpus 2 --mem 4G --disk 5G 22.04
```

3. **Access VM Shell:**

```
multipass shell mininet
```

4. **Install prerequisites inside the VM:**

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y git python3 python3-pip openvswitch-switch tcpdump
```

5. **Install Mininet:**

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

1.3 Topology Implementation Script

`topo.py` defines the network topology as specified in the assignment diagram. It defines 4 hosts (`h1`–`h4`), 4 switches (`s1`–`s4`) and a `dns` resolver. The bandwidth and delay between the links is defined using `self.addLink` in the class.

```python
class AssignmentTopo(Topo):
    def build(self):
        # add the switches
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')
        # add hosts with IPs
        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 = self.addHost('dns', ip='10.0.0.5/24')
        # links
        self.addLink(h1, s1, cls=TCLink, bw=100, delay='2ms')
        self.addLink(h2, s2, cls=TCLink, bw=100, delay='2ms')
        self.addLink(h3, s3, cls=TCLink, bw=100, delay='2ms')
        self.addLink(h4, s4, cls=TCLink, bw=100, delay='2ms')
        # links between switches and dns
        self.addLink(s1, s2, cls=TCLink, bw=100, delay='5ms')
        self.addLink(s2, s3, cls=TCLink, bw=100, delay='8ms')
        self.addLink(s3, s4, cls=TCLink, bw=100, delay='10ms')
        self.addLink(dns, s2, cls=TCLink, bw=100, delay='1ms')
def run():
    topo = AssignmentTopo()
    net = Mininet(topo=topo, controller=Controller, link=TCLink, switch=OVSSwitch, autoSetMacs=True)
    net.start()
    dns = net.get('dns')
    dns.cmd('ip addr add 10.0.2.1/24 dev dns-eth0')
    print("\nNetwork started.\n")
    net.pingAll()
    print("\nStarting Mininet\n")
    CLI(net)
    net.stop()
if __name__ == '__main__':
    run()
```

Listing 1: Python Script: topo.py

1.3.1 Execution

1. **Transfer Script to VM:** using `multipass transfer` command on host machine.

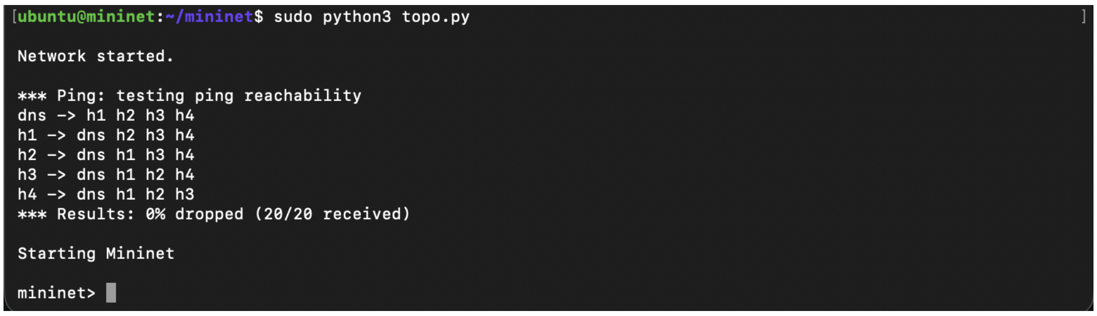
```
multipass transfer topo.py mininet:/home/ubuntu/mininet/topo.py
```

2. **Run the Simulation:** execute script with superuser privileges inside the VM (sudo (s uper u ser do)).

```
sudo python3 ~/mininet/topo.py
```

1.4 Results

The `net.pingAll()` command was used to verify connectivity between all nodes. The output below shows a 0% packet drop rate, confirming that every node can successfully reach every other node in the network.



```
ubuntu@mininet:~/mininet$ sudo python3 topo.py

Network started.

*** 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)

Starting Mininet

mininet> 
```

Figure 1: Results for Task A

2 Task B — DNS Resolution Analysis using Wireshark & Python for Above Mininet Network Topology

2.1 Objective

We have to analyze DNS query performance for multiple hosts in the Mininet network by replaying captured PCAP traces and resolving the extracted domains using the **default host resolver**. The performance metrics are:

- Number of successful and failed DNS resolutions,
- Average lookup latency (ms),
- Average throughput (successful queries per second).

2.2 Methodology

1. **Network Setup:** The hosts `hX` were configured with NAT (Network Address Translation) so they could access external DNS servers non-invasively (done in topo.py):
2. **Data Extraction:** For each host, the corresponding PCAP file (`PCAP_1_H1.pcap` … `PCAP_4_H4.pcap`) contains DNS traffic. Using Wireshark's CLI utility `tshark`, we retrieved a list of unique domain names per host (duplicates/trailing dots removed):

```
tshark -r PCAP_1_H1.pcap -Y "dns && dns.qry.name" \
       -T fields -e dns.qry.name > domains_h1.txt
awk '{gsub(/\.$/,""); print}' domains_h1.txt | awk '!seen[$0]++' \
       > domains_h1_unique.txt
```

3. **DNS Replay Script:** `resolve_list.py` replayed these DNS queries using the system's default resolver via `socket.getaddrinfo()`. Each query was timed and results were written to a CSV file containing:
   - Timestamp
   - Domain name
   - Success (1 = resolved, 0 = failed)
   - Resolution time (ms)
   - Response IP(s)

   The command executed for each host was:

```
sudo python3 resolve_list.py domains_h1_unique.txt \
    h1_dns_results.csv --pause 0.01 --limit 500
```

   The `--pause 0.01` parameter throttles lookups to ∼ 100 queries per second, as suggested by Sameer Sir in class.

2.3 Challenges and Fixes

- **Out-of-memory termination:** Initially, using `rdpcap()` from Scapy caused the process to be killed due to large PCAP files being fully loaded into RAM. *Fix:* Used `tshark` for domain extraction and a streaming resolver (`part_b_resolver.py`) for efficient processing
- **Internet connectivity within Mininet:** Hosts initially lacked external connectivity (`topo_b.py`). *Fix:* Enabled IP forwarding and NAT to allow real DNS queries to reach external resolvers.

2.4 Results and Observations

Each host performed 100 DNS queries (unique domains). The observed results are tabulated below.

Host	Queries	Success	Fail	Success (%)	Avg. Lookup Latency (ms)	Throughput (queries/s)
h1	100	71	29	71%	381.54	1.88
h2	100	68	32	68%	420.47	1.94
h3	100	72	28	72%	336.01	2.34
h4	100	73	27	73%	310.64	2.82

Table 1: DNS resolution performance per host using default resolver.

2.5 Conclusion

The task demonstrates DNS resolution replay in Mininet using real PCAP data and the default DNS resolver, with external connectivity provided through NAT. Results showed variable latency, thruput and success rate across hosts.

3 Task C — Modification of Mininet Topology to use Custom DNS Resolver

3.1 Objective

In this task, we have to modify the existing Mininet topology to use our custom DNS resolver from assignment 1 (instead of the default system resolver used in task B) for DNS resolution.

3.2 Methodology and Implementation

1. **Network Topology:** has 4 hosts (h1-h4), 1 DNS host (`dns` at `10.0.0.5`), 4 switches, and 1 NAT node (`nat0` at `10.0.0.254`) for internet access. After starting, a default route to NAT node was added to all hosts as shown:

```
        # ... network setup ...
        info('Using NAT for external connectivity\n')
        nat = net.addNAT(ip='10.0.0.254/24', inNamespace=False, connect=False)
        net.addLink(nat, s1)

        # ... network start ...
        net.start()
        for host in [h1, h2, h3, h4, dns]:
            host.cmd(f'ip route add default via 10.0.0.254')

```

Listing 2: NAT and default route configuration in `topo_cr.py`

2. **Custom DNS Resolver (`cr.py`:** A Python `socketserver` script was used as a simple DNS forwarder. It binds to `10.0.0.5:53`, listens for queries, and forwards them to an upstream server (`8.8.8.8`). The response is then relayed back to the original host. The resolver was launched on the `dns` host from the main script.

```
        info('*** Launching Custom DNS Resolver on 10.0.0.5\n')
        dns.cmd('sudo python3 cr.py &')
        info('DNS Resolver operational\n')

```

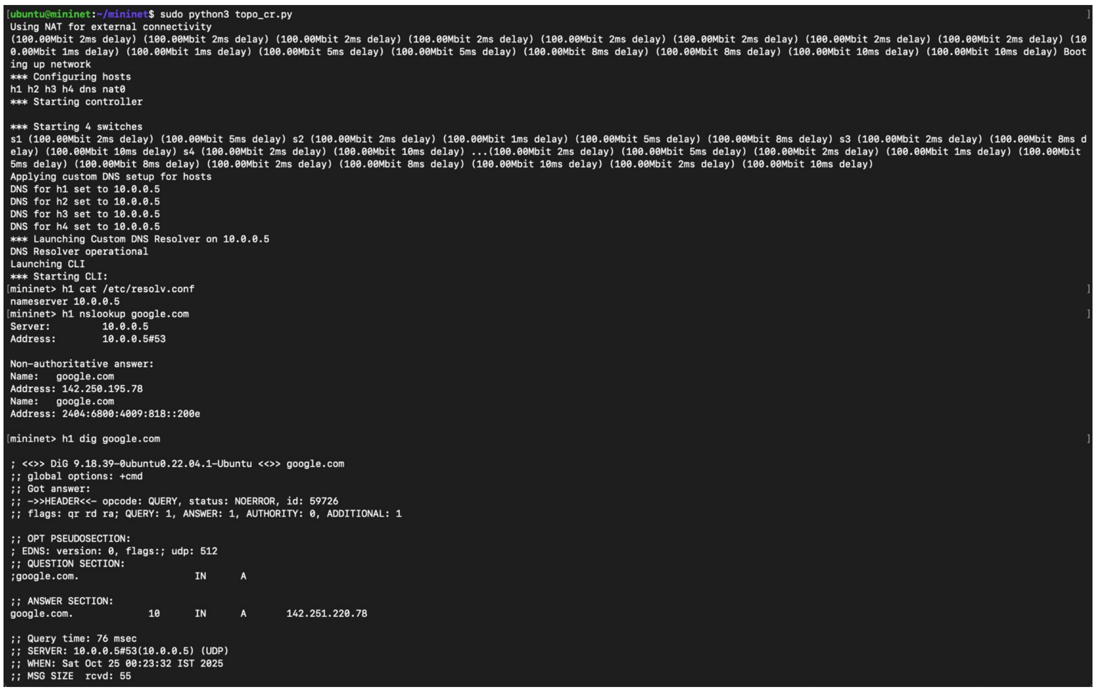
Listing 3: DNS resolver launch in `topo_cr.py`

3. **Modifying Host DNS Configuration:** by overwriting the default `/etc/resolv.conf` on hosts `h1-h4` to remove the inherited configuration and set `nameserver` to `10.0.0.5`, pointing all queries to the custom resolver.

```
    info('Applying custom DNS setup for hosts\n')
    for host in [h1, h2, h3, h4]:
        host.cmd('rm /etc/resolv.conf')
        host.cmd(f'echo "nameserver 10.0.0.5" > /etc/resolv.conf')
        info(f"DNS for {host.name} set to 10.0.0.5\n")
```

Listing 4: Dynamic DNS configuration in `topo_cr.py`

3.3 Results and Verification



```
ubuntu@mininet:~/mininet$ sudo python3 topo_cr.py
Using NAT for external connectivity
(100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (10
0.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mbit 5ms delay) (100.00Mbit 5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay) (100.00Mbit 10ms delay) Boot
ing up network
*** Configuring hosts
h1 h2 h3 h4 dns nat0
*** Starting controller

*** Starting 4 switches
s1 (100.00Mbit 2ms delay) (100.00Mbit 5ms delay) s2 (100.00Mbit 2ms delay) (100.00Mbit 1ms delay) (100.00Mbit 5ms delay) (100.00Mbit 8ms delay) s3 (100.00Mbit 2ms delay) (100.00Mbit 8ms d
elay) (100.00Mbit 10ms delay) s4 (100.00Mbit 2ms delay) (100.00Mbit 10ms delay) ...(100.00Mbit 2ms delay) (100.00Mbit 5ms delay) (100.00Mbit 2ms delay) (100.00Mbit 1ms delay) (100.00Mbit
5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 2ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay) (100.00Mbit 2ms delay) (100.00Mbit 10ms delay)
Applying custom DNS setup for hosts
DNS for h1 set to 10.0.0.5
DNS for h2 set to 10.0.0.5
DNS for h3 set to 10.0.0.5
DNS for h4 set to 10.0.0.5
*** Launching Custom DNS Resolver on 10.0.0.5
DNS Resolver operational
Launching CLI
*** Starting CLI:
mininet> h1 cat /etc/resolv.conf
nameserver 10.0.0.5
mininet> h1 nslookup google.com
Server:         10.0.0.5
Address:        10.0.0.5#53

Non-authoritative answer:
Name:   google.com
Address: 142.250.195.78
Name:   google.com
Address: 2404:6800:4009:818::200e

mininet> h1 dig google.com

; <<>> DiG 9.18.39-0ubuntu0.22.04.1-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 59726
;; 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.             10      IN      A       142.251.220.78

;; Query time: 76 msec
;; SERVER: 10.0.0.5#53(10.0.0.5) (UDP)
;; WHEN: Sat Oct 25 00:23:32 IST 2025
;; MSG SIZE  rcvd: 55
```

Figure 2: Results for part c

The script log shows the DNS server for `h1-h4` was set to `10.0.0.5`. Every `nslookup` test from the hosts confirms the query was handled by `Server:10.0.0.5`, proving the default resolver is bypassed and our custom resolver works.

3.4 Problems Faced and Solutions

A primary challenge was enabling internet access for the Mininet hosts, which is required for the `dns` host to contact the upstream server (`8.8.8.8`). By default, hosts are in an isolated namespace. To solve this, we did the following:

- Using `net.addNAT()` to bridge Mininet network to the host's external interface.
- Adding default route on all hosts (`host.cmd('ip route add default via 10.0.0.254')`) to direct external traffic to NAT node.

Without both, DNS forwarding would fail. Additionally, `sudo` was required to run the script due to privileged operations (binding to port 53, modifying `/etc/resolv.conf`).

3.5 Conclusion

This task showed the working of our custom DNS resolver within Mininet. By modifying the `/etc/resolv.conf` file of each host, all DNS traffic was redirected to a custom Python-based forwarding server on a dedicated host within the topology. `nslookup` confirmed that all external domain queries were correctly handled by our server at `10.0.0.5`.

4 Task D — Custom DNS Resolver Performance Analysis

4.1 Objectives

In this task, we have to:

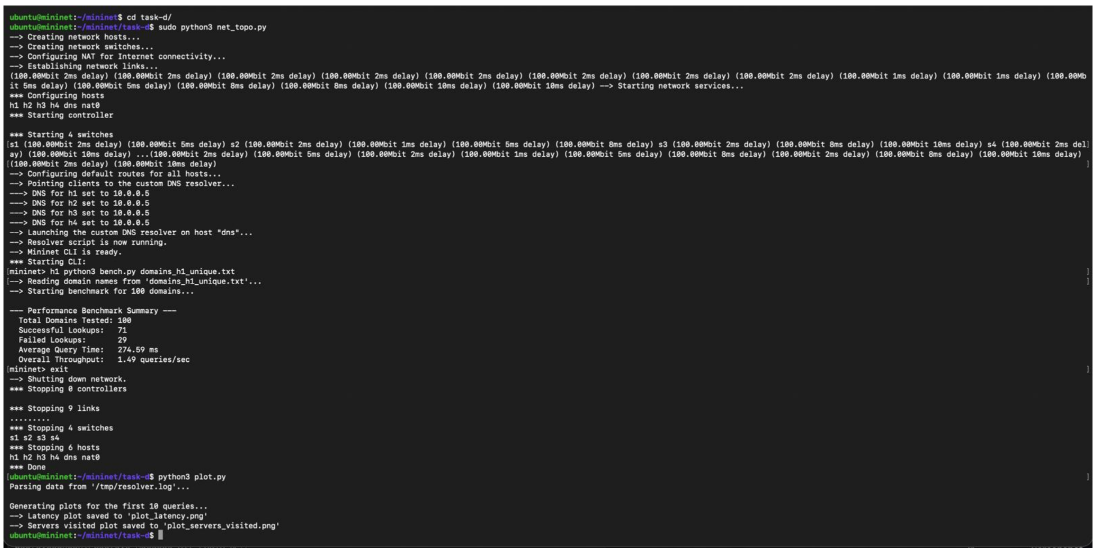
- compare the results of our custom DNS resolver with Mininet's default system resolver based on specific metrics (avg. lookup latency, avg. throughput, no. of successfully resolved queries).
- for each DNS query, log the timestamp, domain name, resolution mode, IP addr. of contacted DNS server, resolution step, response received, RTT, total resolution time, and cache status.
- plot the total number of DNS servers visited and latency for the first 10 URLs queried in `PCAP_1_H1`.

4.2 Methodology

1. **Network Topology Setup using `net_topo.py`:** The `net_topo.py` script (similar to `topo_cr.py` from Part C) establishes a Mininet topology. It creates a controller, switches, a dedicated `dns` host, and client hosts (`h1` - `h4`). It configures all client hosts' `/etc/resolv.conf` to point to the IP address of our custom `dns` host (10.0.0.5). The script also initiates the `cr.py` resolver on the `dns` host, redirecting its output to a log file at `/tmp/resolver.log`.
2. **Custom DNS Resolver using `cr.py`:** This script acts as a DNS forwarding resolver. When it receives query:
   - It extracts the queried domain name.
   - It logs various details (timestamp, domain, resolution mode, etc.) to a file (`/tmp/resolver.log`).
   - It forwards the query directly to a predefined upstream DNS server (e.g., 8.8.8.8).
   - When it receives a response from upstream server, it logs the upstream RTT and the total resolution time.
   - Finally, it sends the response back to the original client host.

   Here, "Resolution Mode" is "Forwarding (Iterative)," and "Cache Status" is "MISS" always. The "Servers Visited" count for any query is always 1, representing the single hop to the upstream public DNS.
3. **Client Benchmark Tool using `bench.py`:**
   - reads pre-extracted domain names from `domains_hX.txt` (instead of using PCAP files).
   - executes the `dig` command, querying the `dns` custom resolver.
   - parses the output of `dig` to extract the query time (latency), throughput, etc. for the benchmark run.
4. **Results Plotting using `plot.py`:**
   - parses the text log file (`/tmp/resolver.log`) generated by the `cr.py` resolver and tshark.
   - extracts domain name and total resolution time for each query, filters to get only the first 10 unique domains.
   - generates 2 plots showing DNS resolution latency (in ms) and no. of upstream DNS servers visited for each of the first 10 unique domains, saving them as `plot_latency.png` and `plot_servers_visited.png`.

4.3 Execution



```
ubuntu@mininet:~/mininet$ cd task-d/
ubuntu@mininet:~/mininet/task-d$ sudo python3 net_topo.py
--> Creating network hosts...
--> Creating network switches...
--> Configuring NAT for Internet connectivity...
--> Establishing network links...
(100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms delay) (100.00Mb
it 5ms delay) (100.00Mbit 5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay) (100.00Mbit 10ms delay) --> Starting network services...
*** Configuring hosts
h1 h2 h3 h4 dns nat0
*** Starting controller

*** Starting 4 switches
s1 (100.00Mbit 2ms delay) (100.00Mbit 5ms delay) s2 (100.00Mbit 2ms delay) (100.00Mbit 1ms delay) (100.00Mbit 5ms delay) (100.00Mbit 8ms delay) s3 (100.00Mbit 2ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay) s4 (100.00Mbit 2ms del
ay) (100.00Mbit 10ms delay) ...(100.00Mbit 2ms delay) (100.00Mbit 5ms delay) (100.00Mbit 2ms delay) (100.00Mbit 1ms delay) (100.00Mbit 5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 2ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay)
(100.00Mbit 2ms delay) (100.00Mbit 10ms delay)
--> Configuring default routes for all hosts...
--> Pointing clients to the custom DNS resolver...
---> DNS for h1 set to 10.0.0.5
---> DNS for h2 set to 10.0.0.5
---> DNS for h3 set to 10.0.0.5
---> DNS for h4 set to 10.0.0.5
--> Launching the custom DNS resolver on host "dns"...
--> Resolver script is now running.
--> Mininet CLI is ready.
*** Starting CLI:
mininet> h1 python3 bench.py domains_h1_unique.txt
--> Reading domain names from 'domains_h1_unique.txt'...
--> Starting benchmark for 100 domains...

--- Performance Benchmark Summary ---
  Total Domains Tested: 100
  Successful Lookups:   71
  Failed Lookups:       29
  Average Query Time:   274.59 ms
  Overall Throughput:   1.49 queries/sec
mininet> exit
--> Shutting down network.
*** Stopping 0 controllers

*** Stopping 9 links
.........
*** Stopping 4 switches
s1 s2 s3 s4
*** Stopping 6 hosts
h1 h2 h3 h4 dns nat0
*** Done
ubuntu@mininet:~/mininet/task-d$ python3 plot.py
Parsing data from '/tmp/resolver.log'...

Generating plots for the first 10 queries...
--> Latency plot saved to 'plot_latency.png'
--> Servers visited plot saved to 'plot_servers_visited.png'
ubuntu@mininet:~/mininet/task-d$
```

1. **Start Mininet and Resolver:** starts network and `cr.py`, redirecting its output to `/tmp/resolver.log`.

```
sudo python3 net_topo.py
```

2. **Run Benchmarks on Hosts:** From the Mininet CLI (repeat for `h2-h4`):

```
mininet> h1 python3 bench.py domains_h1.txt
```

3. **Collect Detailed Resolver Logs:** While still in Mininet (or after exiting), the log can be viewed:

```
mininet> dns cat /tmp/resolver.log
```

4.

```
mininet> exit
```

5. **Generate Plots:** From the regular Linux terminal:

```
$ python3 plot.py
```

4.4 Problems Faced and Solutions

- **Misunderstanding PCAP vs. Text File Input:** *Challenge:* Initially, there was a misunderstanding regarding the direct use of PCAP files. While PCAP files were used in Part B for extracting domains, the `bench.py` script for Part D was designed to accept a simple text file with pre-extracted domain names. *Solution:* Used `tshark` to extract unique domain names from the original PCAP files into `domains_hX.txt` files.
- **Plotting Script File Not Found Error:** *Challenge:* After preparing the plotting script, running it resulted in ‘FileNotFoundError‘ (e.g., `generate_plots_from_log.py` not found). *Solution:* This was a simple human error. The script was saved under the filename ‘plot.py‘ using ‘nano‘, but the execution command inadvertently specified a different (non-existent) filename. Correcting the command to ‘python3 plot.py‘ resolved the issue.
- **Plots Showing Fewer Than 10 Domains:** *Challenge:* Despite ensuring that `domains_h1.txt` contained at least 10 unique domains and running the benchmark, the generated plots displayed only 6 domains instead of the expected 10. *Root Cause Identification:* The `/tmp/resolver.log` file contained multiple entries for the same domain if the `bench.py` was executed multiple times, or if the `dig` command internally retried queries. The original plotting script (`plot.py`) was using `df.head(NUM_URLS_TO_PLOT)`, which took the first 10 rows of the DataFrame regardless of whether the domain names in those rows were unique. *Solution:* The ‘plot.py‘ script was modified to first drop duplicate domain entries before selecting the top 10 for plotting. The line:

```
plot_data = df.head(NUM_URLS_TO_PLOT)
```

  was changed to:

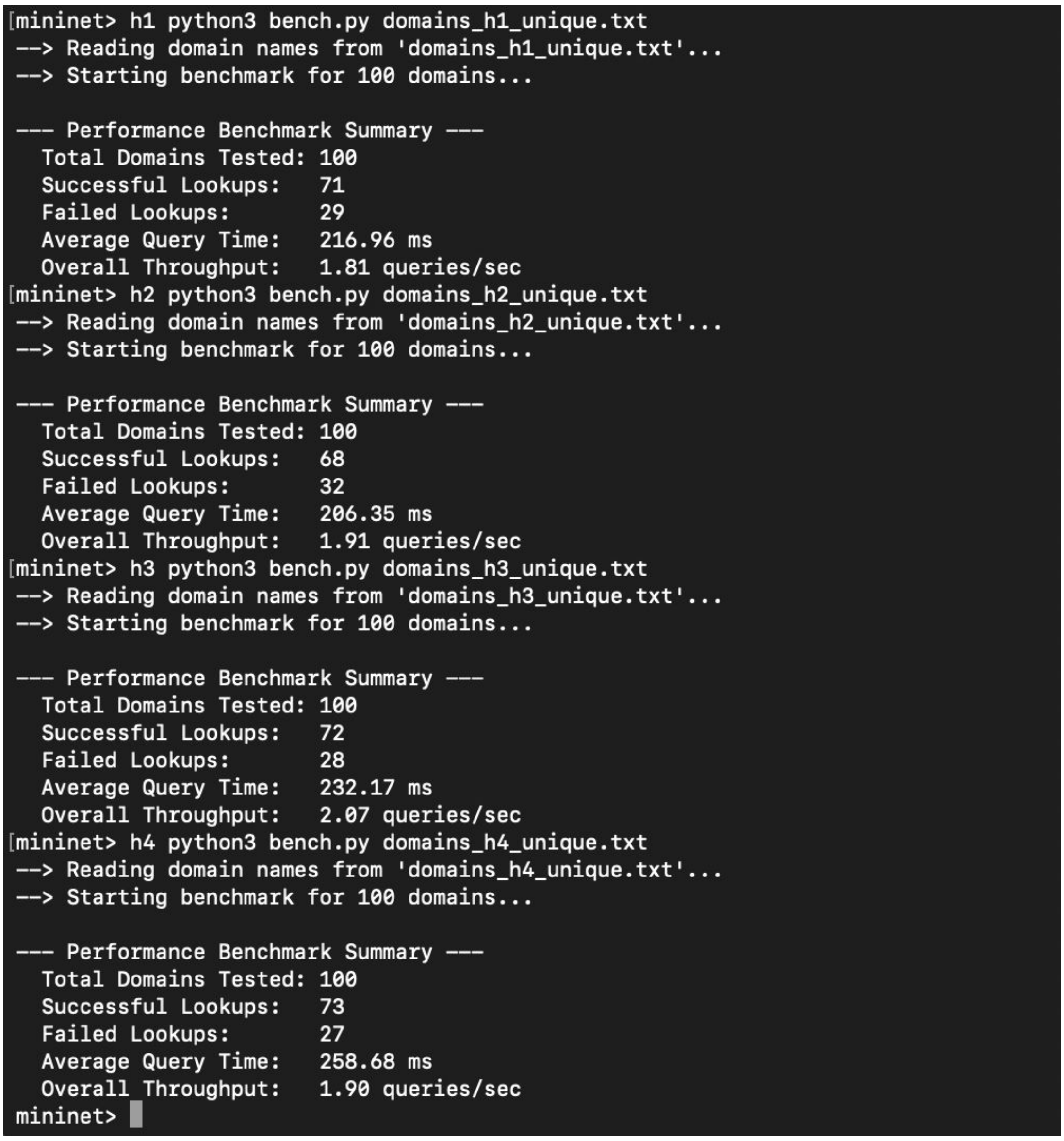
```
plot_data = df.drop_duplicates(subset=['domain']).head(NUM_URLS_TO_PLOT)
```

4.5 Results

- **Detailed Resolver Log (`/tmp/resolver.log`):** An excerpt for a single query looks like this:

```
--- DNS Query Log ---
a. Timestamp: 2025-10-26T20:30:15.123456
b. Domain Name: example.com
c. Resolution Mode: Forwarding (Iterative)
i. Cache Status: MISS
d. DNS Server Contacted: 8.8.8.8
e. Resolution Step: Forwarded to upstream resolver
f. Response Received: Success
g. Upstream RTT: 45.78 ms
h. Overall Time: 50.12 ms
-----------------------
```

- **Comparison of Custom Resolver with Default Resolver:** After running `bench.py` on `hX` with `domains_hX.txt`:



```
[mininet> h1 python3 bench.py domains_h1_unique.txt
 --> Reading domain names from 'domains_h1_unique.txt'...
 --> Starting benchmark for 100 domains...

 --- Performance Benchmark Summary ---
   Total Domains Tested: 100
   Successful Lookups:   71
   Failed Lookups:       29
   Average Query Time:   216.96 ms
   Overall Throughput:   1.81 queries/sec
[mininet> h2 python3 bench.py domains_h2_unique.txt
 --> Reading domain names from 'domains_h2_unique.txt'...
 --> Starting benchmark for 100 domains...

 --- Performance Benchmark Summary ---
   Total Domains Tested: 100
   Successful Lookups:   68
   Failed Lookups:       32
   Average Query Time:   206.35 ms
   Overall Throughput:   1.91 queries/sec
[mininet> h3 python3 bench.py domains_h3_unique.txt
 --> Reading domain names from 'domains_h3_unique.txt'...
 --> Starting benchmark for 100 domains...

 --- Performance Benchmark Summary ---
   Total Domains Tested: 100
   Successful Lookups:   72
   Failed Lookups:       28
   Average Query Time:   232.17 ms
   Overall Throughput:   2.07 queries/sec
[mininet> h4 python3 bench.py domains_h4_unique.txt
 --> Reading domain names from 'domains_h4_unique.txt'...
 --> Starting benchmark for 100 domains...

 --- Performance Benchmark Summary ---
   Total Domains Tested: 100
   Successful Lookups:   73
   Failed Lookups:       27
   Average Query Time:   258.68 ms
   Overall Throughput:   1.90 queries/sec
 mininet> 
```

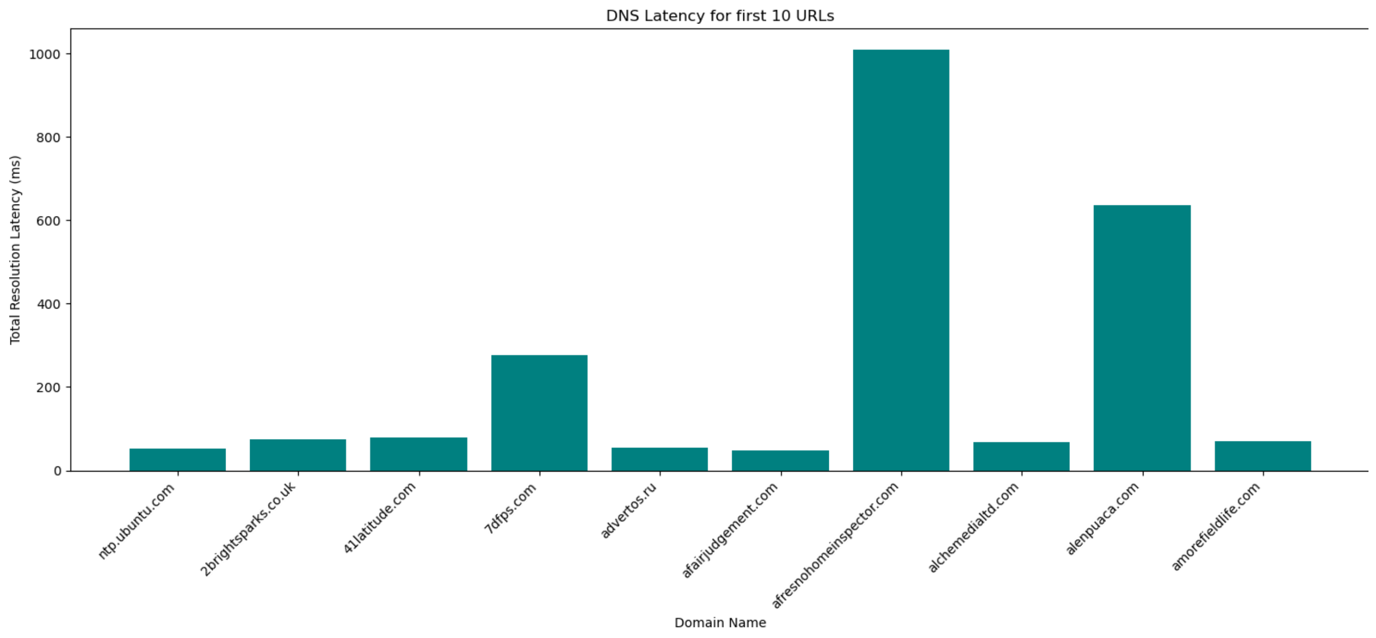
Figure 3: Results for part d

Host	**Queries**	**Success**	**Fail**	**Success (%)**	**Avg. Lookup Latency (ms)**	**Throughput (queries/s)**
h1	100	71	29	71%	216.96	1.81
h2	100	68	32	68%	206.35	1.91
h3	100	72	28	72%	232.17	2.07
h4	100	73	27	73%	258.68	1.90

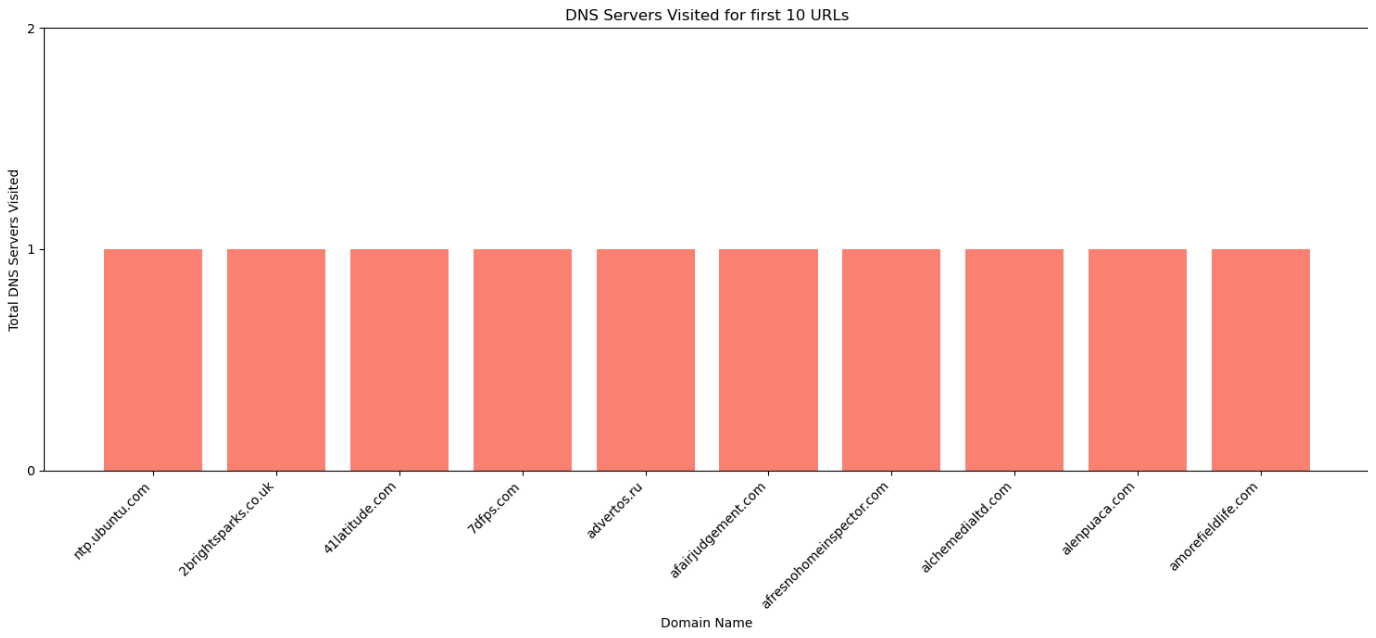
Table 2: DNS resolution performance per host using custom resolver.

Custom has lower avg. latency than default: Default resolver probably retries more aggressively or falls back to TCP on UDP loss. Those retries add 100s of ms to affected queries and inflate avg. latency for default. **Custom has lower avg. throughput than default:** Since custom resolver is single-threaded and limits outstanding queries to 100 q/s, it can't process as many simultaneous queries.

- **Graphical Plots**



The above shows a wide range of resolution times, ($\sim 40ms$ for 'afairjudgement.com' to $> 1000ms$ for 'afresnohomeinspector.com'). This variability is due to factors like the upstream DNS server's load, network congestion b/w Mininet environment and internet, and responsiveness of authoritative DNS servers for each domain.



The above shows only 1 server for all queries because the custom resolver is a forwarding resolver: it does not perform recursive lookups itself by querying root, TLD, and authoritative servers; it sends every received query to the upstream server. Thus, from the perspective of `cr.py` resolver, only 1 external server is visited per query.