CS331: Computer Networks Assignment 2

Mamilla Aniruddh Reddy (23110195) Ogiboyina Akash (23110225)

Task: DNS Query Resolution

Subtask-A: Simulating a topology in Mininet

H1(10.0.0.1) <--BW=100 Mbps, Delay=2ms-->S1

H2(10.0.0.2) <--BW=100 Mbps, Delay=2ms-->S2

H3(10.0.0.3) <--BW=100Mbps, Delay=2ms-->S3

H4(10.0.0.4) <--BW=100 Mbps, Delay=2ms-->S4

S2<--BW=100 Mbps, Delay=1ms-->DNS Resolver (10.0.0.5)

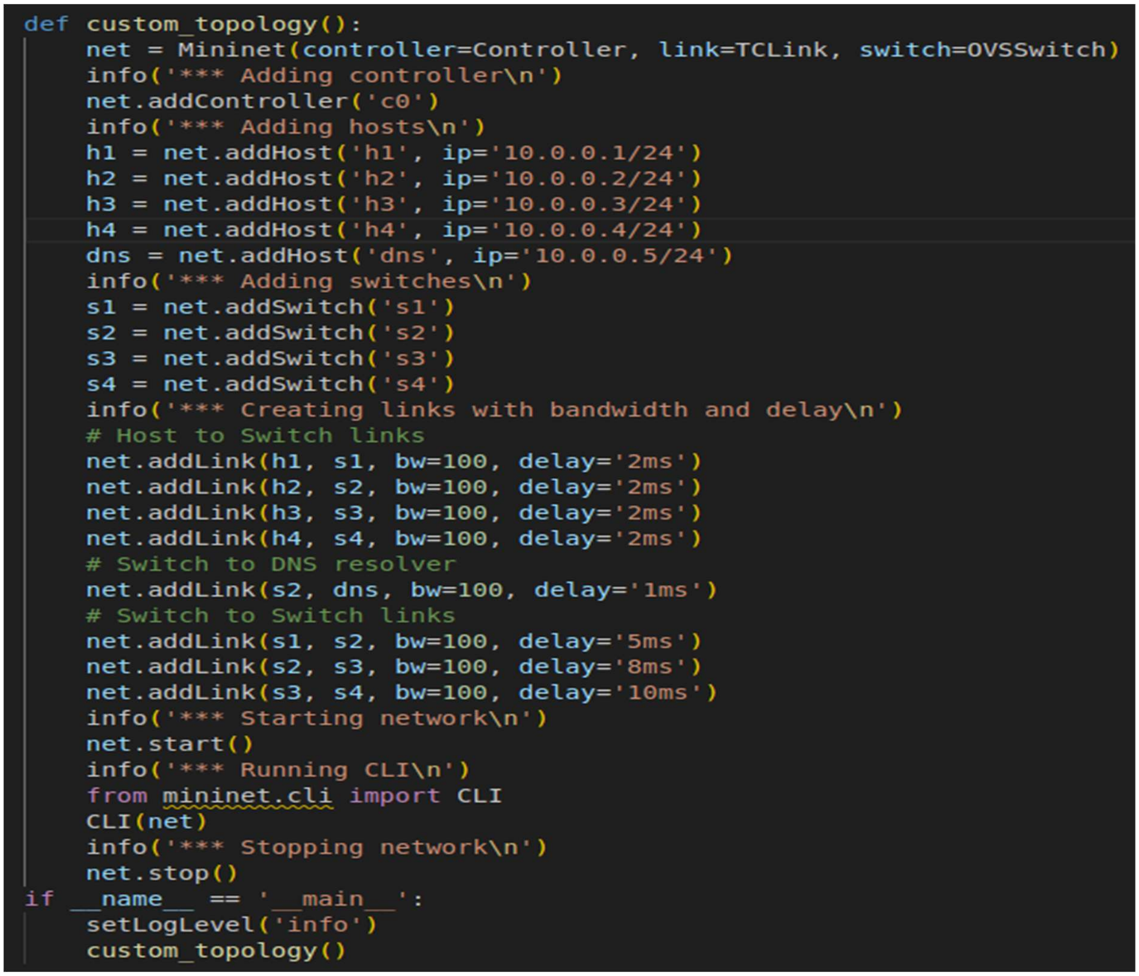
S1<--BW=100 Mbps, Delay=5ms-->S2

S2<--BW=100 Mbps, Delay=8ms-->S3

S3<--BW=100 Mbps, Delay=10ms-->S4

Where H=Host, S=Switch and BW=Band Width

Made a Python script “dns_topology.py” for simulating the given topology in Mininet and demonstrating successful connectivity among all nodes.

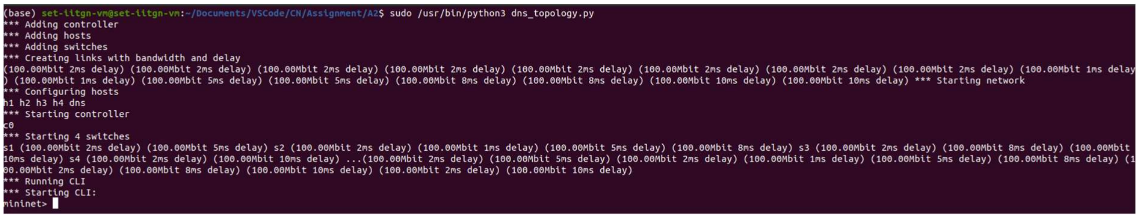


```
def custom_topology():
    net = Mininet(controller=Controller, link=TCLink, switch=OVSSwitch)
    info('*** Adding controller\n')
    net.addController('c0')
    info('*** Adding hosts\n')
    h1 = net.addHost('h1', ip='10.0.0.1/24')
    h2 = net.addHost('h2', ip='10.0.0.2/24')
    h3 = net.addHost('h3', ip='10.0.0.3/24')
    h4 = net.addHost('h4', ip='10.0.0.4/24')
    dns = net.addHost('dns', ip='10.0.0.5/24')
    info('*** Adding switches\n')
    s1 = net.addSwitch('s1')
    s2 = net.addSwitch('s2')
    s3 = net.addSwitch('s3')
    s4 = net.addSwitch('s4')
    info('*** Creating links with bandwidth and delay\n')
    # Host to Switch links
    net.addLink(h1, s1, bw=100, delay='2ms')
    net.addLink(h2, s2, bw=100, delay='2ms')
    net.addLink(h3, s3, bw=100, delay='2ms')
    net.addLink(h4, s4, bw=100, delay='2ms')
    # Switch to DNS resolver
    net.addLink(s2, dns, bw=100, delay='1ms')
    # Switch to Switch links
    net.addLink(s1, s2, bw=100, delay='5ms')
    net.addLink(s2, s3, bw=100, delay='8ms')
    net.addLink(s3, s4, bw=100, delay='10ms')
    info('*** Starting network\n')
    net.start()
    info('*** Running CLI\n')
    from mininet.cli import CLI
    CLI(net)
    info('*** Stopping network\n')
    net.stop()
if __name__ == '__main__':
    setLogLevel('info')
    custom_topology()
```

CS331: Computer Networks Assignment 1

The custom_topology() function creates four hosts (H1–H4), a DNS resolver, and four interconnected switches (S1–S4) with specified bandwidths and delays. Each host is connected to its respective switch, forming the exact network given in the problem statement. Finally, it launches the Mininet CLI for interactive testing and verification of network connectivity.

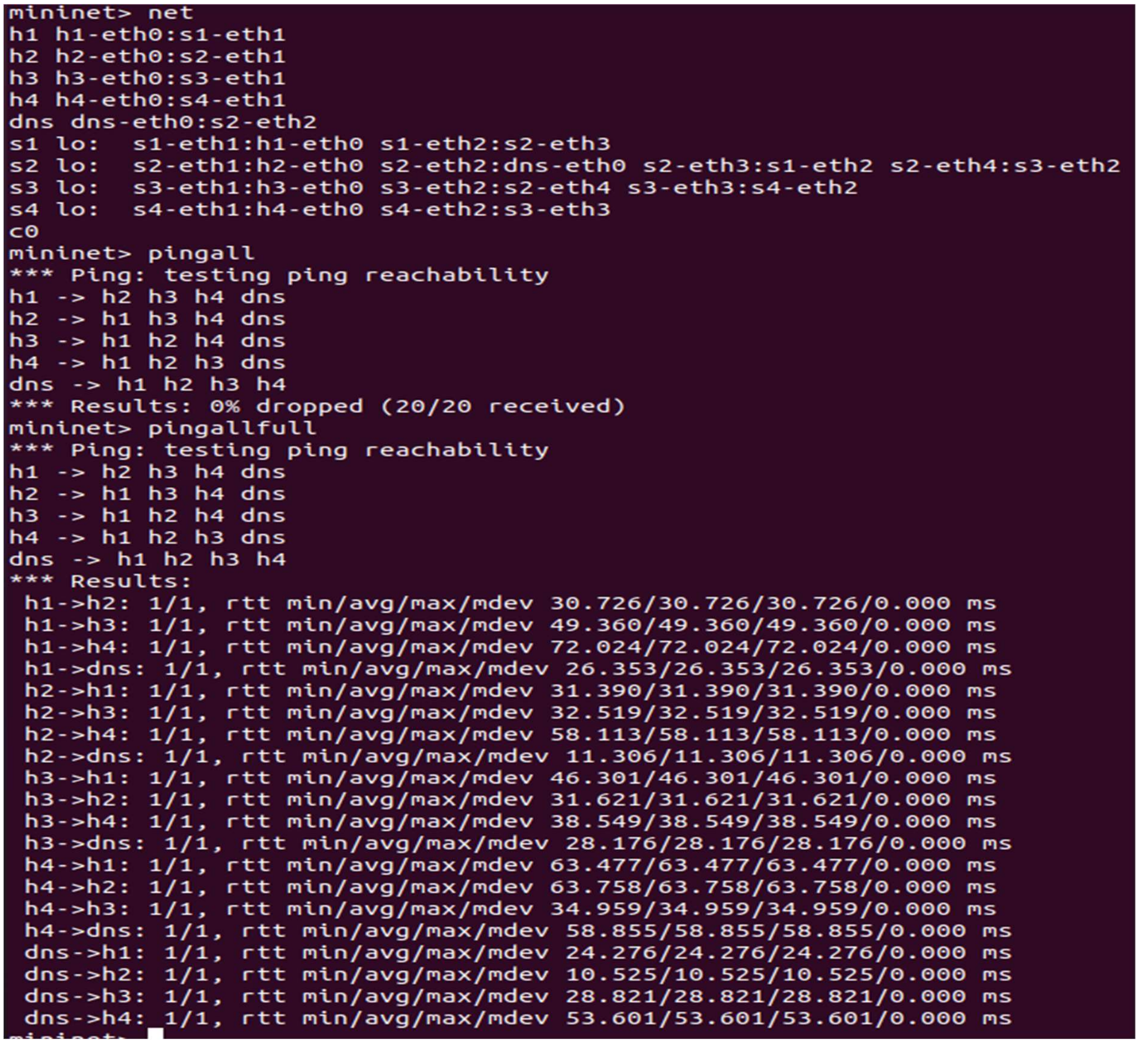
The output of the script confirms our simulation matches the given topology in Mininet.



```
(base) set-iitgn-vm@set-iitgn-vm:~/Documents/VSCode/CN/Assignment/A2$ sudo /usr/bin/python3 dns_topology.py
*** Adding controller
*** Adding hosts
*** Adding switches
*** Creating links with bandwidth and 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 2ms delay) (100.00Mbit 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) *** Starting network
*** Configuring hosts
h1 h2 h3 h4 dns
*** Starting controller
c0
*** 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 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) (1
00.00Mbit 2ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay) (100.00Mbit 2ms delay) (100.00Mbit 10ms delay)
*** Running CLI
*** Starting CLI:
mininet>
```

We can demonstrate successful connectivity using some Mininet commands.

Net: This command lists the nodes in the network.
pingall: This command pings between all hosts in the network to test connectivity.
pingallfull: It pings between all hosts in the network and reports the results of each ping.



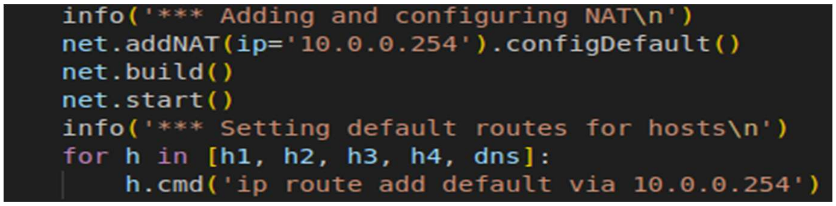
```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
h4 h4-eth0:s4-eth1
dns dns-eth0:s2-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth3
s2 lo:  s2-eth1:h2-eth0 s2-eth2:dns-eth0 s2-eth3:s1-eth2 s2-eth4:s3-eth2
s3 lo:  s3-eth1:h3-eth0 s3-eth2:s2-eth4 s3-eth3:s4-eth2
s4 lo:  s4-eth1:h4-eth0 s4-eth2:s3-eth3
c0
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 dns
h2 -> h1 h3 h4 dns
h3 -> h1 h2 h4 dns
h4 -> h1 h2 h3 dns
dns -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
mininet> pingallfull
*** Ping: testing ping reachability
h1 -> h2 h3 h4 dns
h2 -> h1 h3 h4 dns
h3 -> h1 h2 h4 dns
h4 -> h1 h2 h3 dns
dns -> h1 h2 h3 h4
*** Results:
 h1->h2: 1/1, rtt min/avg/max/mdev 30.726/30.726/30.726/0.000 ms
 h1->h3: 1/1, rtt min/avg/max/mdev 49.360/49.360/49.360/0.000 ms
 h1->h4: 1/1, rtt min/avg/max/mdev 72.024/72.024/72.024/0.000 ms
 h1->dns: 1/1, rtt min/avg/max/mdev 26.353/26.353/26.353/0.000 ms
 h2->h1: 1/1, rtt min/avg/max/mdev 31.390/31.390/31.390/0.000 ms
 h2->h3: 1/1, rtt min/avg/max/mdev 32.519/32.519/32.519/0.000 ms
 h2->h4: 1/1, rtt min/avg/max/mdev 58.113/58.113/58.113/0.000 ms
 h2->dns: 1/1, rtt min/avg/max/mdev 11.306/11.306/11.306/0.000 ms
 h3->h1: 1/1, rtt min/avg/max/mdev 46.301/46.301/46.301/0.000 ms
 h3->h2: 1/1, rtt min/avg/max/mdev 31.621/31.621/31.621/0.000 ms
 h3->h4: 1/1, rtt min/avg/max/mdev 38.549/38.549/38.549/0.000 ms
 h3->dns: 1/1, rtt min/avg/max/mdev 28.176/28.176/28.176/0.000 ms
 h4->h1: 1/1, rtt min/avg/max/mdev 63.477/63.477/63.477/0.000 ms
 h4->h2: 1/1, rtt min/avg/max/mdev 63.758/63.758/63.758/0.000 ms
 h4->h3: 1/1, rtt min/avg/max/mdev 34.959/34.959/34.959/0.000 ms
 h4->dns: 1/1, rtt min/avg/max/mdev 58.855/58.855/58.855/0.000 ms
 dns->h1: 1/1, rtt min/avg/max/mdev 24.276/24.276/24.276/0.000 ms
 dns->h2: 1/1, rtt min/avg/max/mdev 10.525/10.525/10.525/0.000 ms
 dns->h3: 1/1, rtt min/avg/max/mdev 28.821/28.821/28.821/0.000 ms
 dns->h4: 1/1, rtt min/avg/max/mdev 53.601/53.601/53.601/0.000 ms
mininet>
```

CS331: Computer Networks Assignment 1

Subtask-B:

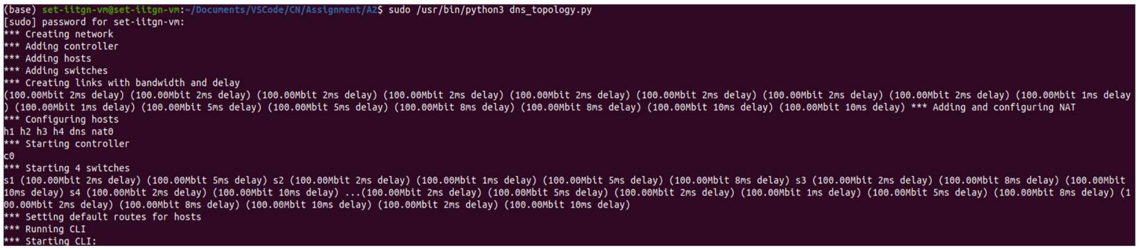
Resolving URLs from the host’s PCAP files using the default host resolver

The dns_topology.py script is modified so that all hosts (H1–H4) and the DNS Resolver can reach the Internet via a NAT host. The added code block sets up a NAT gateway and assigns default routes to enable connectivity.



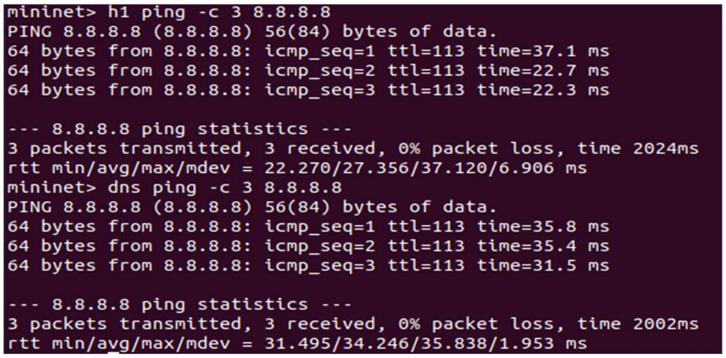
```
    info('*** Adding and configuring NAT\n')
    net.addNAT(ip='10.0.0.254').configDefault()
    net.build()
    net.start()
    info('*** Setting default routes for hosts\n')
    for h in [h1, h2, h3, h4, dns]:
        h.cmd('ip route add default via 10.0.0.254')
```

When we rerun the script, we can observe the addition of the NAT node.



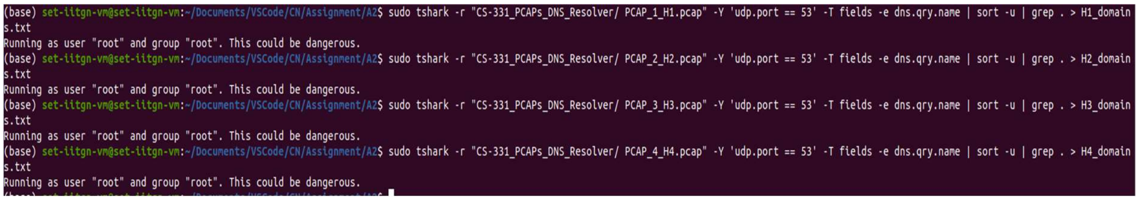
```
(base) set-iitgn-vm@set-iitgn-vm:~/Documents/VSCode/CN/Assignment/A2$ sudo /usr/bin/python3 dns_topology.py
[sudo] password for set-iitgn-vm:
*** Creating network
*** Adding controller
*** Adding hosts
*** Adding switches
*** Creating links with bandwidth and delay
(100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 2ms delay) (100.00Mbit 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) *** Adding and configuring NAT
*** Configuring hosts
h1 h2 h3 h4 dns nat0
*** Starting controller
c0
*** 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 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) (1
00.00Mbit 2ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay) (100.00Mbit 2ms delay) (100.00Mbit 10ms delay)
*** Setting default routes for hosts
*** Running CLI
*** Starting CLI:
```

We can ping any host and the DNS Resolver to send ICMP echo requests to check internet connectivity.



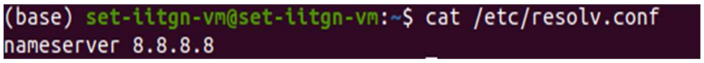
```
mininet> h1 ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=37.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=22.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=22.3 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2024ms
rtt min/avg/max/mdev = 22.270/27.356/37.120/6.906 ms
mininet> dns ping -c 3 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=113 time=35.8 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=35.4 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=31.5 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 31.495/34.246/35.838/1.953 ms
```

We used tshark to read the PCAP files of each host and extracted all unique DNS query domain names sent over UDP port 53. The results are filtered, sorted, and saved into text files of format “Hi_domains.txt”.



```
(base) set-iitgn-vm@set-iitgn-vm:~/Documents/VSCode/CN/Assignment/A2$ sudo tshark -r "CS-331_PCAPs_DNS_Resolver/ PCAP_1_H1.pcap" -Y 'udp.port == 53' -T fields -e dns.qry.name | sort -u | grep . > H1_domain
s.txt
Running as user "root" and group "root". This could be dangerous.
(base) set-iitgn-vm@set-iitgn-vm:~/Documents/VSCode/CN/Assignment/A2$ sudo tshark -r "CS-331_PCAPs_DNS_Resolver/ PCAP_2_H2.pcap" -Y 'udp.port == 53' -T fields -e dns.qry.name | sort -u | grep . > H2_domain
s.txt
Running as user "root" and group "root". This could be dangerous.
(base) set-iitgn-vm@set-iitgn-vm:~/Documents/VSCode/CN/Assignment/A2$ sudo tshark -r "CS-331_PCAPs_DNS_Resolver/ PCAP_3_H3.pcap" -Y 'udp.port == 53' -T fields -e dns.qry.name | sort -u | grep . > H3_domain
s.txt
Running as user "root" and group "root". This could be dangerous.
(base) set-iitgn-vm@set-iitgn-vm:~/Documents/VSCode/CN/Assignment/A2$ sudo tshark -r "CS-331_PCAPs_DNS_Resolver/ PCAP_4_H4.pcap" -Y 'udp.port == 53' -T fields -e dns.qry.name | sort -u | grep . > H4_domain
s.txt
Running as user "root" and group "root". This could be dangerous.
```

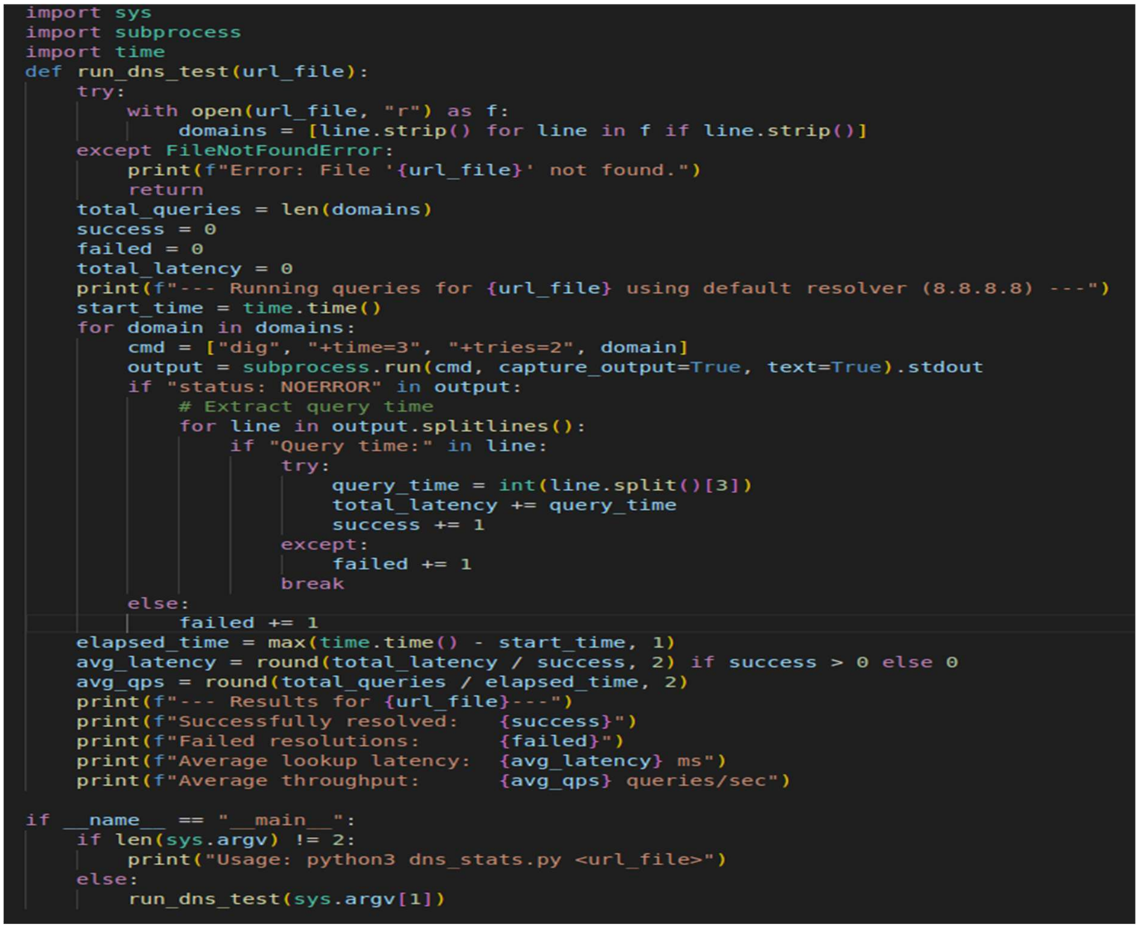
Checking the configuration of the default DNS resolver.



```
(base) set-iitgn-vm@set-iitgn-vm:~$ cat /etc/resolv.conf
nameserver 8.8.8.8
```

CS331: Computer Networks Assignment 1

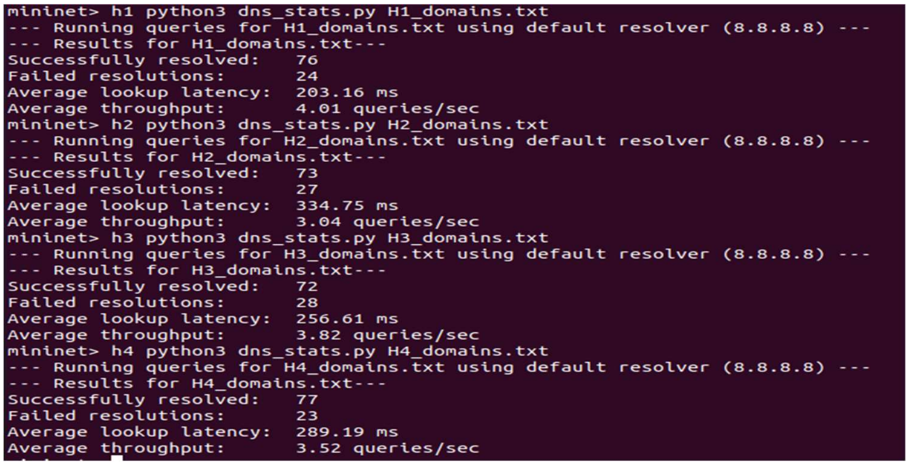
Made a Python script “dns_stats.py” to resolve the URL using the default resolver and measure DNS performance metrics like average lookup latency, throughput, and query success rates.



```python
import sys
import subprocess
import time
def run_dns_test(url_file):
    try:
        with open(url_file, "r") as f:
            domains = [line.strip() for line in f if line.strip()]
    except FileNotFoundError:
        print(f"Error: File '{url_file}' not found.")
        return
    total_queries = len(domains)
    success = 0
    failed = 0
    total_latency = 0
    print(f"--- Running queries for {url_file} using default resolver (8.8.8.8) ---")
    start_time = time.time()
    for domain in domains:
        cmd = ["dig", "+time=3", "+tries=2", domain]
        output = subprocess.run(cmd, capture_output=True, text=True).stdout
        if "status: NOERROR" in output:
            # Extract query time
            for line in output.splitlines():
                if "Query time:" in line:
                    try:
                        query_time = int(line.split()[3])
                        total_latency += query_time
                        success += 1
                    except:
                        failed += 1
                    break
        else:
            failed += 1
    elapsed_time = max(time.time() - start_time, 1)
    avg_latency = round(total_latency / success, 2) if success > 0 else 0
    avg_qps = round(total_queries / elapsed_time, 2)
    print(f"--- Results for {url_file}---")
    print(f"Successfully resolved:   {success}")
    print(f"Failed resolutions:      {failed}")
    print(f"Average lookup latency:  {avg_latency} ms")
    print(f"Average throughput:      {avg_qps} queries/sec")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: python3 dns_stats.py <url_file>")
    else:
        run_dns_test(sys.argv[1])
```

The function run_dns_test() reads domain names from a file (extracted from PCAPs) and runs DNS lookups using the dig command. It records the number of successful and failed resolutions, as well as total query time. Average latency and throughput are computed to evaluate resolver efficiency.

Now, for each host, we run the script on each host along with the respective text file containing the domain names.



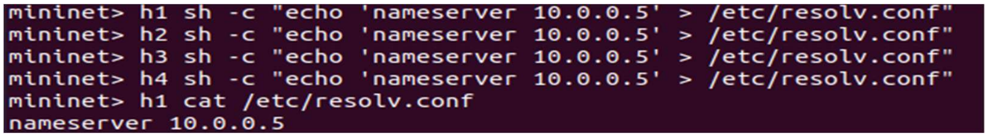
```
mininet> h1 python3 dns_stats.py H1_domains.txt
--- Running queries for H1_domains.txt using default resolver (8.8.8.8) ---
--- Results for H1_domains.txt---
Successfully resolved:   76
Failed resolutions:      24
Average lookup latency:  203.16 ms
Average throughput:      4.01 queries/sec
mininet> h2 python3 dns_stats.py H2_domains.txt
--- Running queries for H2_domains.txt using default resolver (8.8.8.8) ---
--- Results for H2_domains.txt---
Successfully resolved:   73
Failed resolutions:      27
Average lookup latency:  334.75 ms
Average throughput:      3.04 queries/sec
mininet> h3 python3 dns_stats.py H3_domains.txt
--- Running queries for H3_domains.txt using default resolver (8.8.8.8) ---
--- Results for H3_domains.txt---
Successfully resolved:   72
Failed resolutions:      28
Average lookup latency:  256.61 ms
Average throughput:      3.82 queries/sec
mininet> h4 python3 dns_stats.py H4_domains.txt
--- Running queries for H4_domains.txt using default resolver (8.8.8.8) ---
--- Results for H4_domains.txt---
Successfully resolved:   77
Failed resolutions:      23
Average lookup latency:  289.19 ms
Average throughput:      3.52 queries/sec
```

CS331: Computer Networks Assignment 1

Subtask-C:

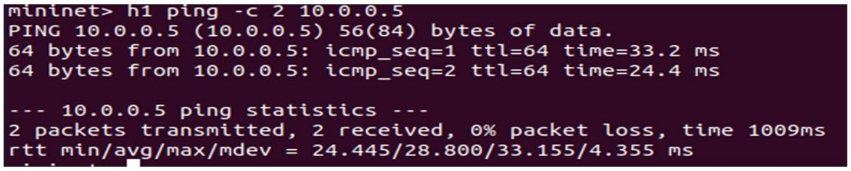
Modifying the DNS configuration of all Mininet hosts to use our custom resolver as the primary DNS server instead of the default system resolver.

Updated the DNS configuration of all the hosts in Mininet to use the custom DNS resolver (10.0.0.5) as its primary nameserver. It ensures that all DNS queries from hi are directed to the custom resolver instead of the default system DNS.



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

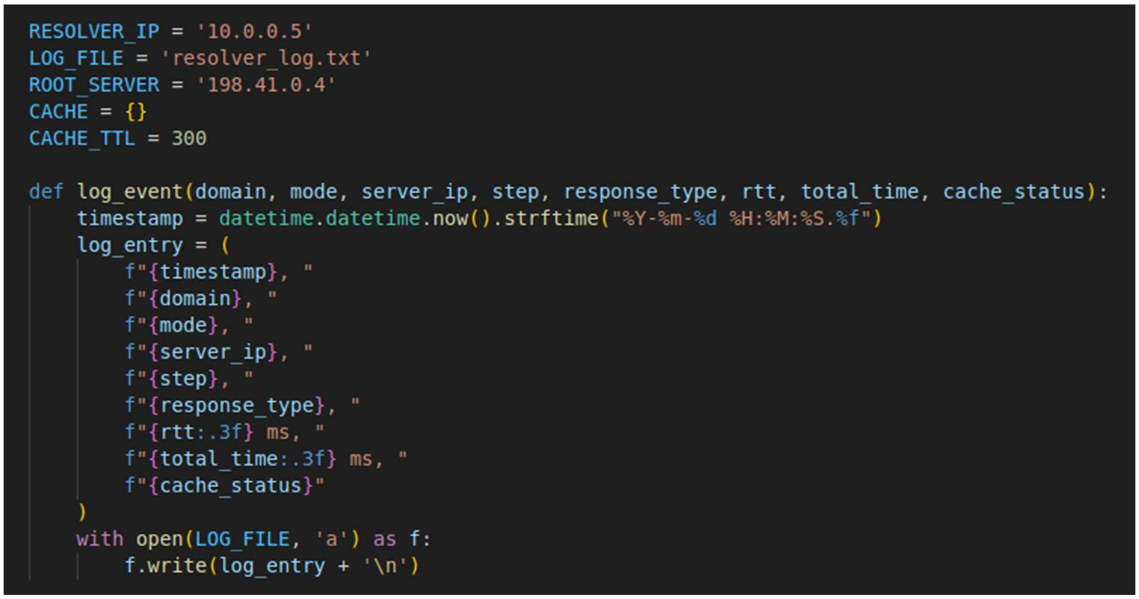
We ping any host to send ICMP echo requests from host Hi to the DNS resolver at 10.0.0.5 to verify connectivity. The reply confirms that the network link between h1 and the DNS server is correctly configured and operational.



```
mininet> h1 ping -c 2 10.0.0.5
PING 10.0.0.5 (10.0.0.5) 56(84) bytes of data.
64 bytes from 10.0.0.5: icmp_seq=1 ttl=64 time=33.2 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=24.4 ms

--- 10.0.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1009ms
rtt min/avg/max/mdev = 24.445/28.800/33.155/4.355 ms
```

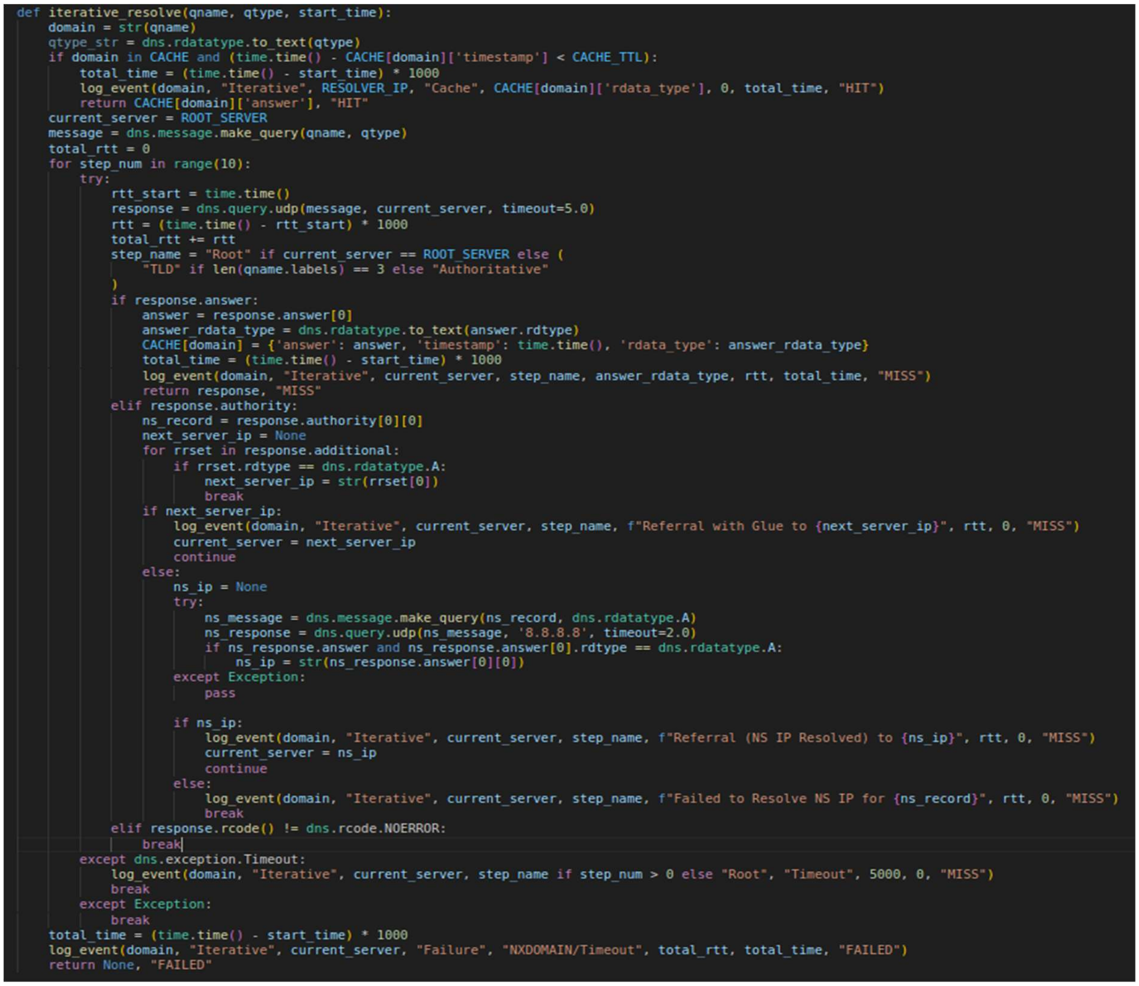
Subtask-D:

Made a Python script “custom_dns_resolver.py” that implements a custom iterative DNS resolver (10.0.0.5).



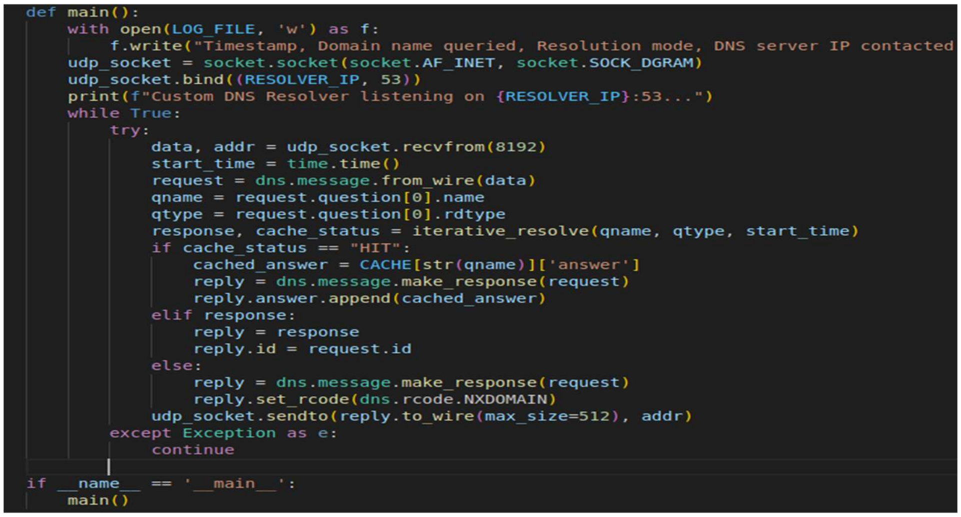
```python
RESOLVER_IP = '10.0.0.5'
LOG_FILE = 'resolver_log.txt'
ROOT_SERVER = '198.41.0.4'
CACHE = {}
CACHE_TTL = 300

def log_event(domain, mode, server_ip, step, response_type, rtt, total_time, cache_status):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")
    log_entry = (
        f"{timestamp}, "
        f"{domain}, "
        f"{mode}, "
        f"{server_ip}, "
        f"{step}, "
        f"{response_type}, "
        f"{rtt:.3f} ms, "
        f"{total_time:.3f} ms, "
        f"{cache_status}"
    )
    with open(LOG_FILE, 'a') as f:
        f.write(log_entry + '\n')
```

The log_event() function records detailed information about each DNS query, including Timestamp, Domain name queried, Resolution mode, DNS server IP contacted, Step of resolution (Root/TLD/Authoritative/Cache), Response or referral received, RTT to that server, Total time to resolution, Cache status (HIT/MISS) into a log file named resolve_log.txt.



```python
def iterative_resolve(qname, qtype, start_time):
    domain = str(qname)
    qtype_str = dns.rdatatype.to_text(qtype)
    if domain in CACHE and (time.time() - CACHE[domain]['timestamp'] < CACHE_TTL):
        total_time = (time.time() - start_time) * 1000
        log_event(domain, "Iterative", RESOLVER_IP, "Cache", CACHE[domain]['rdata_type'], 0, total_time, "HIT")
        return CACHE[domain]['answer'], "HIT"
    current_server = ROOT_SERVER
    message = dns.message.make_query(qname, qtype)
    total_rtt = 0
    for step_num in range(10):
        try:
            rtt_start = time.time()
            response = dns.query.udp(message, current_server, timeout=5.0)
            rtt = (time.time() - rtt_start) * 1000
            total_rtt += rtt
            step_name = "Root" if current_server == ROOT_SERVER else (
                "TLD" if len(qname.labels) == 3 else "Authoritative"
            )
            if response.answer:
                answer = response.answer[0]
                answer_rdata_type = dns.rdatatype.to_text(answer.rdtype)
                CACHE[domain] = {'answer': answer, 'timestamp': time.time(), 'rdata_type': answer_rdata_type}
                total_time = (time.time() - start_time) * 1000
                log_event(domain, "Iterative", current_server, step_name, answer_rdata_type, rtt, total_time, "MISS")
                return response, "MISS"
            elif response.authority:
                ns_record = response.authority[0][0]
                next_server_ip = None
                for rrset in response.additional:
                    if rrset.rdtype == dns.rdatatype.A:
                        next_server_ip = str(rrset[0])
                        break
                if next_server_ip:
                    log_event(domain, "Iterative", current_server, step_name, f"Referral with Glue to {next_server_ip}", rtt, 0, "MISS")
                    current_server = next_server_ip
                    continue
                else:
                    ns_ip = None
                    try:
                        ns_message = dns.message.make_query(ns_record, dns.rdatatype.A)
                        ns_response = dns.query.udp(ns_message, '8.8.8.8', timeout=2.0)
                        if ns_response.answer and ns_response.answer[0].rdtype == dns.rdatatype.A:
                            ns_ip = str(ns_response.answer[0][0])
                    except Exception:
                        pass

                    if ns_ip:
                        log_event(domain, "Iterative", current_server, step_name, f"Referral (NS IP Resolved) to {ns_ip}", rtt, 0, "MISS")
                        current_server = ns_ip
                        continue
                    else:
                        log_event(domain, "Iterative", current_server, step_name, f"Failed to Resolve NS IP for {ns_record}", rtt, 0, "MISS")
                        break
            elif response.rcode() != dns.rcode.NOERROR:
                break
        except dns.exception.Timeout:
            log_event(domain, "Iterative", current_server, step_name if step_num > 0 else "Root", "Timeout", 5000, 0, "MISS")
            break
        except Exception:
            break
    total_time = (time.time() - start_time) * 1000
    log_event(domain, "Iterative", current_server, "Failure", "NXDOMAIN/Timeout", total_rtt, total_time, "FAILED")
    return None, "FAILED"
```

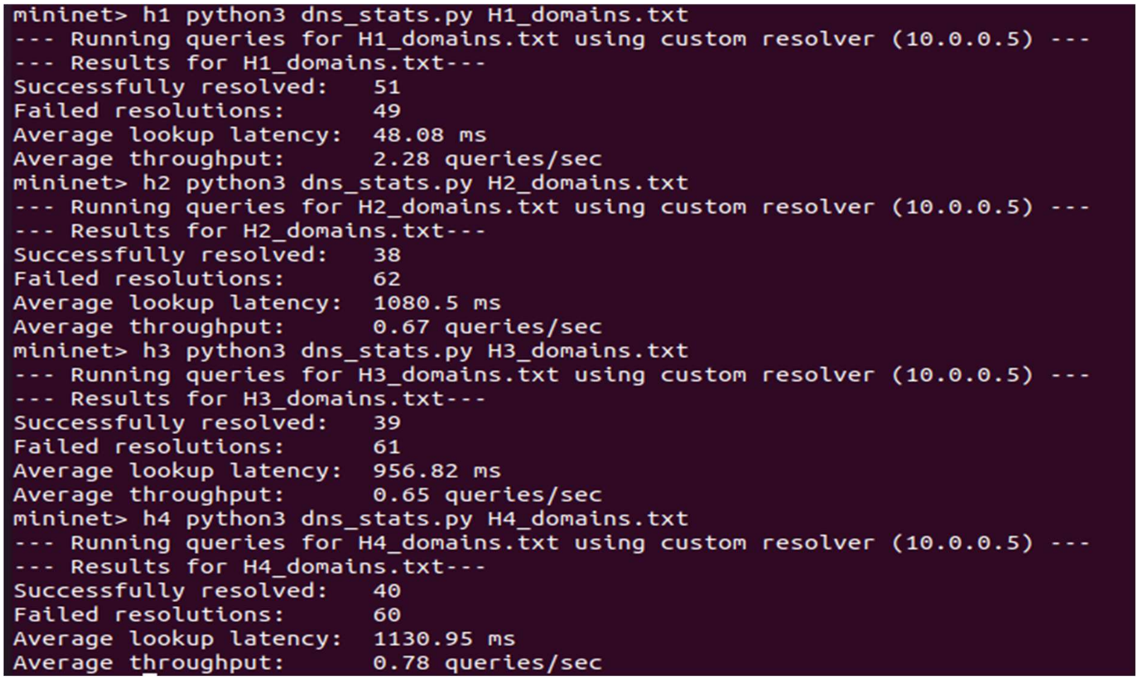
The iterative_resolve() function performs the actual iterative DNS resolution, starting from the root server and following referrals until it reaches the authoritative server, caching responses along the way to speed up future lookups. It also handles DNS timeouts, missing glue records, and failed name resolutions.



```python
def main():
    with open(LOG_FILE, 'w') as f:
        f.write("Timestamp, Domain name queried, Resolution mode, DNS server IP contacted
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_socket.bind((RESOLVER_IP, 53))
    print(f"Custom DNS Resolver listening on {RESOLVER_IP}:53...")
    while True:
        try:
            data, addr = udp_socket.recvfrom(8192)
            start_time = time.time()
            request = dns.message.from_wire(data)
            qname = request.question[0].name
            qtype = request.question[0].rdtype
            response, cache_status = iterative_resolve(qname, qtype, start_time)
            if cache_status == "HIT":
                cached_answer = CACHE[str(qname)]['answer']
                reply = dns.message.make_response(request)
                reply.answer.append(cached_answer)
            elif response:
                reply = response
                reply.id = request.id
            else:
                reply = dns.message.make_response(request)
                reply.set_rcode(dns.rcode.NXDOMAIN)
            udp_socket.sendto(reply.to_wire(max_size=512), addr)
        except Exception as e:
            continue

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

The script initializes the resolver by binding a UDP socket to port 53, listens for incoming DNS queries, and responds either from the cache or through iterative resolution.

We now run the “dns_stats.py” script again for each host, along with the respective text file containing the domain names, using our custom resolver as the primary DNS server.



```
mininet> h1 python3 dns_stats.py H1_domains.txt
--- Running queries for H1_domains.txt using custom resolver (10.0.0.5) ---
--- Results for H1_domains.txt---
Successfully resolved:   51
Failed resolutions:      49
Average lookup latency:  48.08 ms
Average throughput:      2.28 queries/sec
mininet> h2 python3 dns_stats.py H2_domains.txt
--- Running queries for H2_domains.txt using custom resolver (10.0.0.5) ---
--- Results for H2_domains.txt---
Successfully resolved:   38
Failed resolutions:      62
Average lookup latency:  1080.5 ms
Average throughput:      0.67 queries/sec
mininet> h3 python3 dns_stats.py H3_domains.txt
--- Running queries for H3_domains.txt using custom resolver (10.0.0.5) ---
--- Results for H3_domains.txt---
Successfully resolved:   39
Failed resolutions:      61
Average lookup latency:  956.82 ms
Average throughput:      0.65 queries/sec
mininet> h4 python3 dns_stats.py H4_domains.txt
--- Running queries for H4_domains.txt using custom resolver (10.0.0.5) ---
--- Results for H4_domains.txt---
Successfully resolved:   40
Failed resolutions:      60
Average lookup latency:  1130.95 ms
Average throughput:      0.78 queries/sec
```

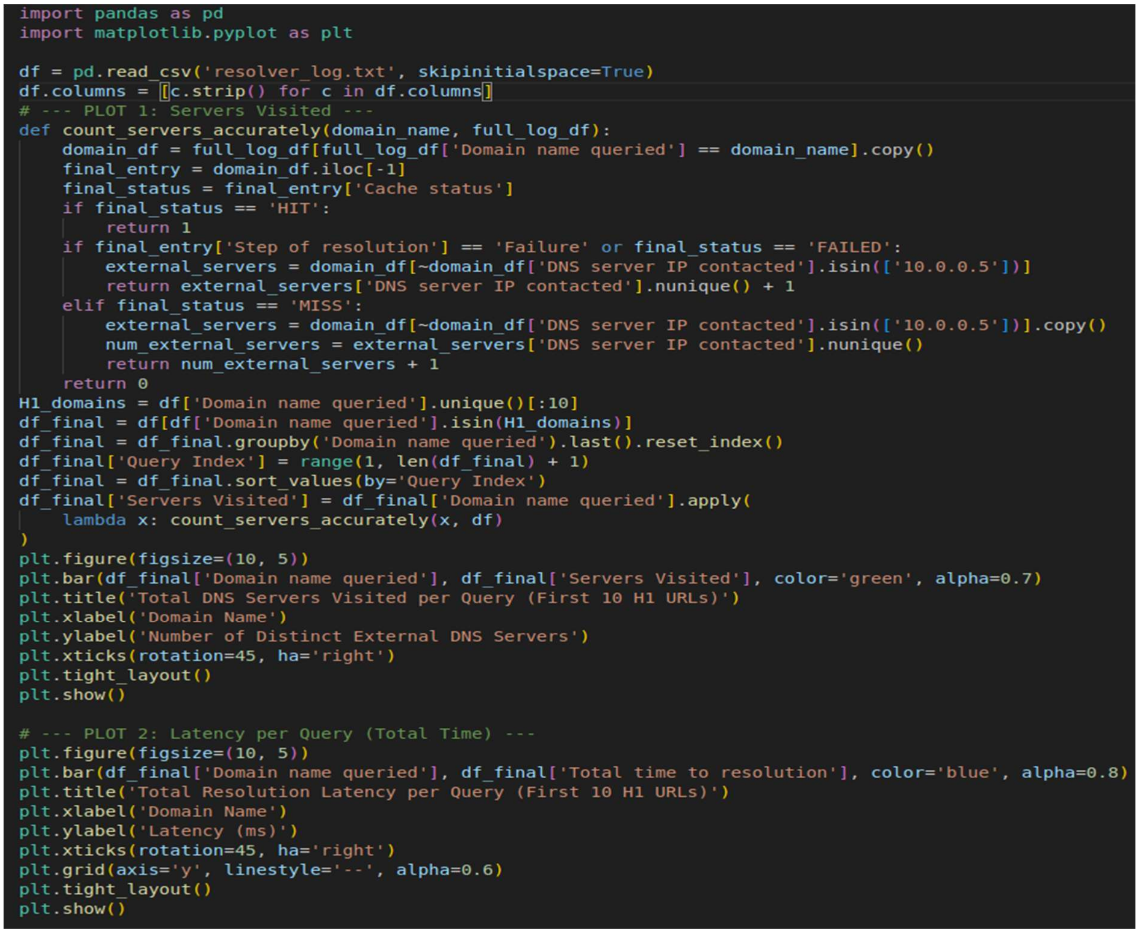
Comparison Table

Host	Resolver	Average Latency (ms)	Average Throughput (q/s)	Success count	Failure count
H1	Default	203.16	4.01	76	24
H1	Custom	48.08	2.28	51	49
H2	Default	334.75	3.04	73	27
H2	Custom	1080.5	0.67	38	62
H3	Default	256.61	3.82	72	28
H3	Custom	956.82	0.65	39	61
H4	Default	289.19	3.52	77	23
H4	Custom	1130.95	0.78	40	60

The results show that the default system resolver generally achieved lower latency and higher throughput across all hosts compared to the custom resolver. While the custom resolver provided detailed logging and caching capabilities, its iterative query process led to higher latency and more failures due to timeouts or incomplete resolutions. The average latency increased significantly for hosts H2–H4 under the custom resolver, indicating longer multi-step lookups. Success rates were also lower for the custom resolver, as it relied entirely on direct DNS traversal without external fallback. Moreover, based on the topology, all DNS queries from the hosts were routed through switch S2, which connects directly to the custom DNS server. This network structure introduced additional propagation delays and potential congestion at S2, as all host requests converged there. Consequently, the topology itself likely contributed to the higher latency and lower throughput observed with the custom resolver.

CS331: Computer Networks Assignment 1

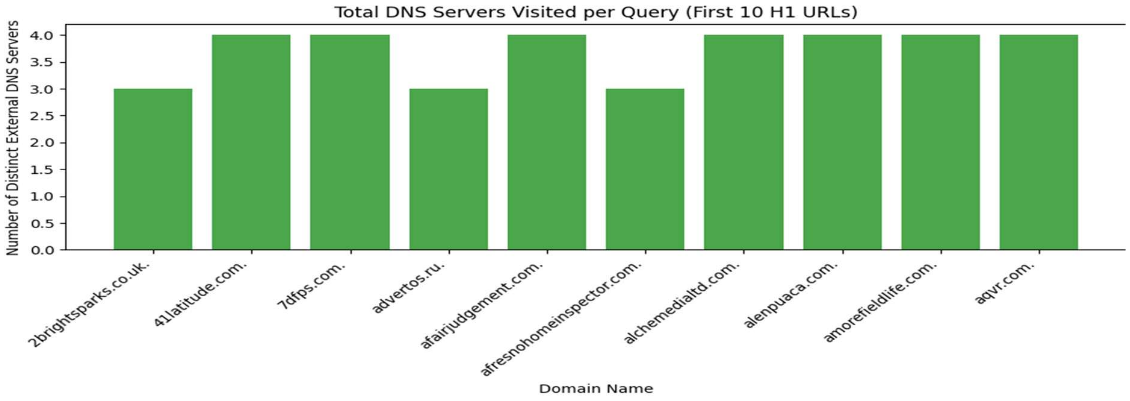
Made a Python script “Plots.py” for presenting graphical plots for the first 10 URLs in PCAP_1_H1. The first plot shows the total number of distinct DNS servers visited per domain, illustrating the number of hops required for each query. The second plot displays the total resolution latency for the exact 10 domains, highlighting variations in query response times.



```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('resolver_log.txt', skipinitialspace=True)
df.columns = [c.strip() for c in df.columns]
# --- PLOT 1: Servers Visited ---
def count_servers_accurately(domain_name, full_log_df):
    domain_df = full_log_df[full_log_df['Domain name queried'] == domain_name].copy()
    final_entry = domain_df.iloc[-1]
    final_status = final_entry['Cache status']
    if final_status == 'HIT':
        return 1
    if final_entry['Step of resolution'] == 'Failure' or final_status == 'FAILED':
        external_servers = domain_df[~domain_df['DNS server IP contacted'].isin(['10.0.0.5'])]
        return external_servers['DNS server IP contacted'].nunique() + 1
    elif final_status == 'MISS':
        external_servers = domain_df[~domain_df['DNS server IP contacted'].isin(['10.0.0.5'])].copy()
        num_external_servers = external_servers['DNS server IP contacted'].nunique()
        return num_external_servers + 1
    return 0
H1_domains = df['Domain name queried'].unique()[:10]
df_final = df[df['Domain name queried'].isin(H1_domains)]
df_final = df_final.groupby('Domain name queried').last().reset_index()
df_final['Query Index'] = range(1, len(df_final) + 1)
df_final = df_final.sort_values(by='Query Index')
df_final['Servers Visited'] = df_final['Domain name queried'].apply(
    lambda x: count_servers_accurately(x, df)
)
plt.figure(figsize=(10, 5))
plt.bar(df_final['Domain name queried'], df_final['Servers Visited'], color='green', alpha=0.7)
plt.title('Total DNS Servers Visited per Query (First 10 H1 URLs)')
plt.xlabel('Domain Name')
plt.ylabel('Number of Distinct External DNS Servers')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# --- PLOT 2: Latency per Query (Total Time) ---
plt.figure(figsize=(10, 5))
plt.bar(df_final['Domain name queried'], df_final['Total time to resolution'], color='blue', alpha=0.8)
plt.title('Total Resolution Latency per Query (First 10 H1 URLs)')
plt.xlabel('Domain Name')
plt.ylabel('Latency (ms)')
plt.xticks(rotation=45, ha='right')
plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

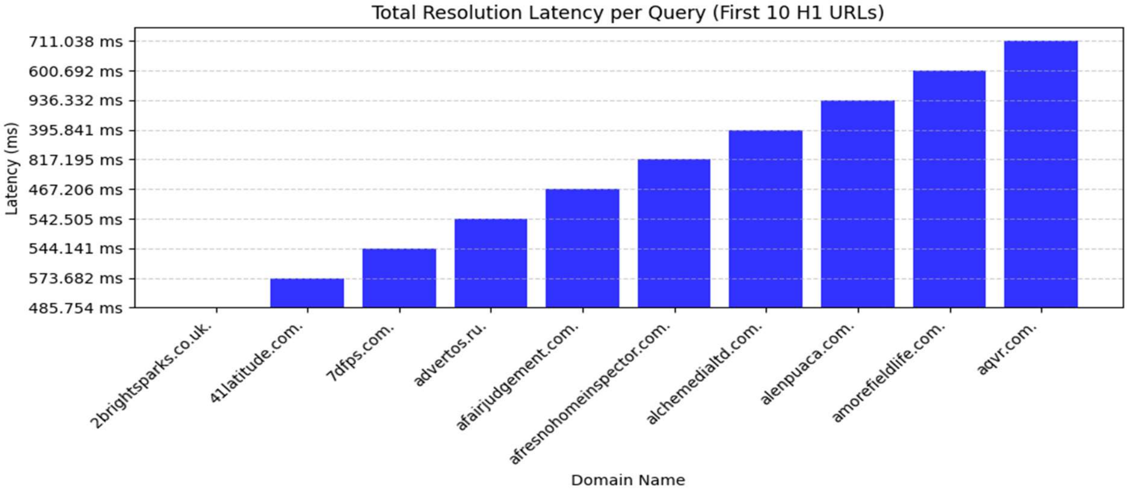
The count_servers_accurately() function calculates the number of distinct external DNS servers contacted during the resolution of each domain, taking into account cache hits, failures, and iterative lookups to root, TLD, and authoritative servers. It helps quantify the depth of the DNS resolution process for each query.

Plot-1: Number of Distinct External DNS Servers vs Domain names



The bar plot analysis reveals that most resolutions (7 out of 10) required contacting four distinct servers (1 Resolver + 3 external servers), reflecting the standard iterative path of Root to TLD to Authoritative. The remaining three domains completed the process by contacting only three distinct servers, likely because the needed information was contained earlier in the referral chain or the resolution failed before querying the third external server, accurately mapping the complexity of the iterative lookup process. There were no hits in the first 10 domains, so we could not see the value of the number of distinct external DNS servers as 1.

Plot-2: Latency per query (ms) vs Domain names



This bar plot analysis showed that the latency, which ranges from approximately 486 ms to over 711 ms, is dominated by the cumulative Round-Trip Time (RTT) incurred across the multiple hops of the iterative path (Root, TLD, Authoritative servers), combined with the non-zero link delays configured in the Mininet topology. The wide variation in latency across the 10 domains reflects differences in network congestion, the number of resolution steps required to find the nameserver IP addresses, and whether the final server responded quickly or slowly.