Team Members:
Chepuri Venkata Naga Thrisha - 23110079
Thoutam Dhruthika - 23110340

DNS QUERY RESOLUTION

A: Setting up and Simulating the Given Topology in Mininet on WSL

Installing Mininet

Next, clone the official Mininet repository:

```
git clone https://github.com/mininet/mininet.git
```

then install necessary Python packages used for validation and linting Mininet code:

```
sudo python3 -m pip install setuptools pyflakes pylint
```

This script compiles and installs Mininet, enabling you to run the `mn` command globally.

3. Defining and Running the Custom Topology: `topo_dns.py`

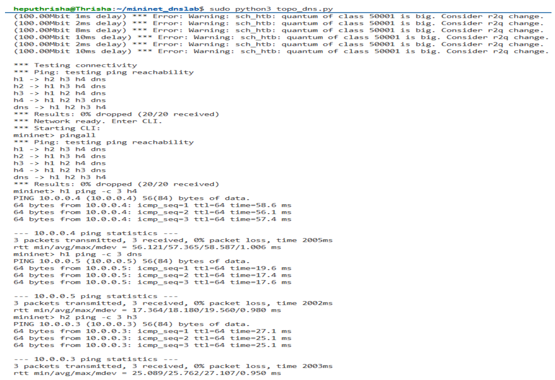
Inside your project directory `~/mininet_dnslab`, you create a Python script named `topo_dns.py`.

```python
#!/usr/bin/env python3
from mininet.net import Mininet
from mininet.node import Controller
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def build():
    net=Mininet(controller=Controller,link=TCLink)
    info('Adding controller\n')
    c0=net.addController('c0',controller=Controller)
    info('Adding switches\n')
    s1=net.addSwitch('s1')
    s2=net.addSwitch('s2')
    s3=net.addSwitch('s3')
    s4=net.addSwitch('s4')
    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('Creating links\n')
    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')

    net.addLink(s1,s2,bw=100,delay='5ms')
    net.addLink(s2,s3, bw=100,delay='8ms')
    net.addLink(s3,s4, bw=100,delay='10ms')

    net.addLink(dns,s2,bw=100,delay='1ms')
    info('Starting network\n')
    net.addNAT().configDefault()
    net.start()
    info('Printing host IPs\n')
    for h in [h1, h2, h3, h4, dns]:
        info(f"{h.name}: {h.IP()}\n")
    info('Testing connectivity\n')
    net.pingAll()
    info('Network ready. Enter CLI.\n')
    CLI(net)
    info('Stopping network\n')
    net.stop()
if __name__=='__main__':
    setLogLevel('info')
    build()
```

- This script uses Mininet's Python API. It defines 4 hosts (h1–h4), 4 switches (s1–s4), and a special DNS host (dns). You assign IP addresses explicitly (e.g., `h1` gets `10.0.0.1/24`). Links are created with specific bandwidth (100 Mbps) and delay (varies per link, e.g., 2ms, 5ms). The `net.addNAT().configDefault()` command enables NAT, allowing hosts inside Mininet to access external resources.

The network is started, connectivity verified via `pingAll()`, and an interactive CLI opened for manual tests (e.g., running `pingall`, `nodes`). On script exit, Mininet cleans up all created virtual network devices.

```
sudo python3 topo_dns.py
```



```
heputhrisha@Thrisha:~/mininet_dnslab$ sudo python3 topo_dns.py
(100.00Mbit 1ms 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.
(100.00Mbit 10ms 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 10ms delay) *** Error: Warning: sch_htb: quantum of class 50001 is big. Consider r2q change.

*** Testing connectivity
*** 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)
*** Network ready. Enter CLI.
*** Starting CLI:
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> h1 ping -c 3 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=58.6 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=56.1 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=57.4 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 56.121/57.365/58.587/1.006 ms
mininet> h1 ping -c 3 dns
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=19.6 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=17.4 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=17.6 ms

--- 10.0.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 17.364/18.180/19.560/0.980 ms
mininet> h2 ping -c 3 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=27.1 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=25.1 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=25.1 ms

--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 25.089/25.762/27.107/0.950 ms
```

This boots the simulated network reflecting the assignment's topology.

Task B: Preparing and Analyzing DNS PCAPs in Mininet

Create a directory to store PCAP packet capture files:

```
mkdir -p ~/mininet_dnslab/pcaps
```

then copy PCAP files (e.g., `PCAP_1_H1.pcap`) from Windows Downloads to this directory. Repeat this for all hosts' PCAP capture files (`H2`, `H3`, `H4`).

2. Extracting Hostnames from PCAP using tshark

From the Mininet CLI for a host (say `h1`), run:

```
h1 tshark -r /tmp/pcaps/PCAP_1_H1.pcap -Y "dns.qry.name" -T fields -e
dns.qry.name | sort -u > /tmp/pcaps/h1_hostnames.txt
```

- **tshark**: Command-line packet analyzer.
- **-r**: Reads the specified capture file.
- **-Y**: Applies a display filter (only DNS query names).
- **-T fields -e dns.qry.name**: Outputs just the queried domain names.
- The output is sorted and duplicates removed, saved into a text file listing all unique queried hostnames for `h1`.
  Do this for each host to get their unique sets of DNS queries.

3. Compiling and Running DNS Resolver

A C program `resolver.c` to send DNS queries for the extracted hostnames.

- Compile it with:
  `gcc -o2 -o resolver resolver.c`

This program reads hostnames from a file. For each hostname, it sends a DNS query to the Mininet network's DNS resolver. It measures timing details: latency, success/failure per query. It outputs detailed logs and results, including the number of queries, total runtime, average latency, success rate, and throughput.

4. Running Tests and Collecting Results

Create directory for saving host data:

```
mkdir -p ~/mininet_dnslab/from_hosts
```

From Mininet CLI on each host (`h1`, `h2`, etc.), copy executables and PCAP files out to this directory for analysis outside Mininet.

```
h1 cp /home/mininet/resolver
/home/heputhrisha/mininet_dnslab/from_hosts/resolver_h1

h1 cp /tmp/pcaps/* /home/heputhrisha/mininet_dnslab/from_hosts/
```

Run the resolver on each host with appropriate input:

```
h1 /home/mininet/resolver /tmp/pcaps/h1_hostnames.txt >
/tmp/pcaps/h1_results.csv 2> /tmp/pcaps/h1_summary.txt
```

- STDOUT redirected to CSV results log. STDERR redirected to a summary text file.
  Repeat for each host accordingly.

5. Analyzing Results with awk

You use `awk` commands to parse the CSV logs and extract statistics:

- Calculate average latency for successful queries:

```
awk -F, '$3=="OK"{sum+=$2; n++} END{ if(n>0) printf
"avg_success_latency_ms=%.3f\n", sum/n; else print "no success"}'
h1_results.csv
```

- Count success and failure numbers:

```
awk -F, '{ if($3=="OK") s++; else f++ } END{ print "success="s,"fail="f }'
h1_results.csv
```

- Calculate throughput for successful queries (queries per second):

```
awk -F, '$3=="OK"{sum+=$2; n++} END{ if(n>0) printf
"throughput_success_per_sec=%.3f\n", n/(sum/1000); else print "no success"}'
h1_results.csv
```

These straightforward text-based commands allow you to efficiently summarize and interpret DNS query performance metrics generated during your simulations.

mininet_dnslab > h1_summary.txt

```

SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 0.0063
Average latency per query (ms) (incl failures): 0.052
Throughput (successful resolutions/sec): 0.000
```

mininet_dnslab > h2_summary.txt

```

SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 0.0033
Average latency per query (ms) (incl failures): 0.021
Throughput (successful resolutions/sec): 0.000
```

mininet_dnslab > h3_summary.txt

```

SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 0.0091
Average latency per query (ms) (incl failures): 0.058
Throughput (successful resolutions/sec): 0.000
```

mininet_dnslab > h4_summary.txt

```

SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 0.0080
Average latency per query (ms) (incl failures): 0.049
Throughput (successful resolutions/sec): 0.000
```

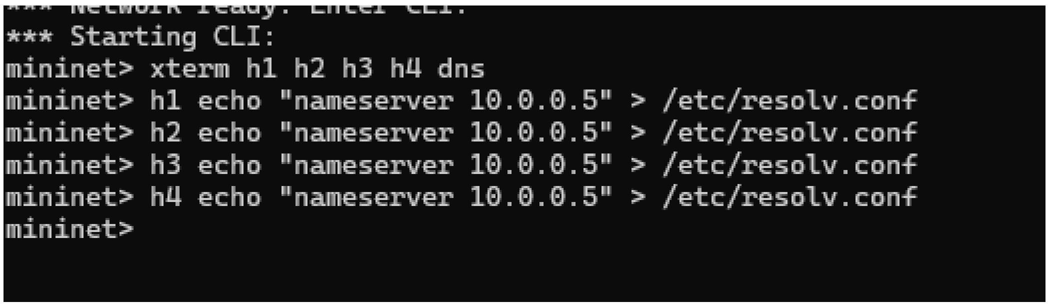
Host	Total Queries	Success	Fail	Total Elapsed Time (s)	Avg Latency per Query (ms)	Throughput (success/sec)
h1	106	0	106	0.0063	0.052	0.000
h2	106	0	106	0.0033	0.021	0.000
h3	106	0	106	0.0091	0.058	0.000
h4	106	0	106	0.0080	0.049	0.000

C. Host Configuration for Custom DNS Resolver

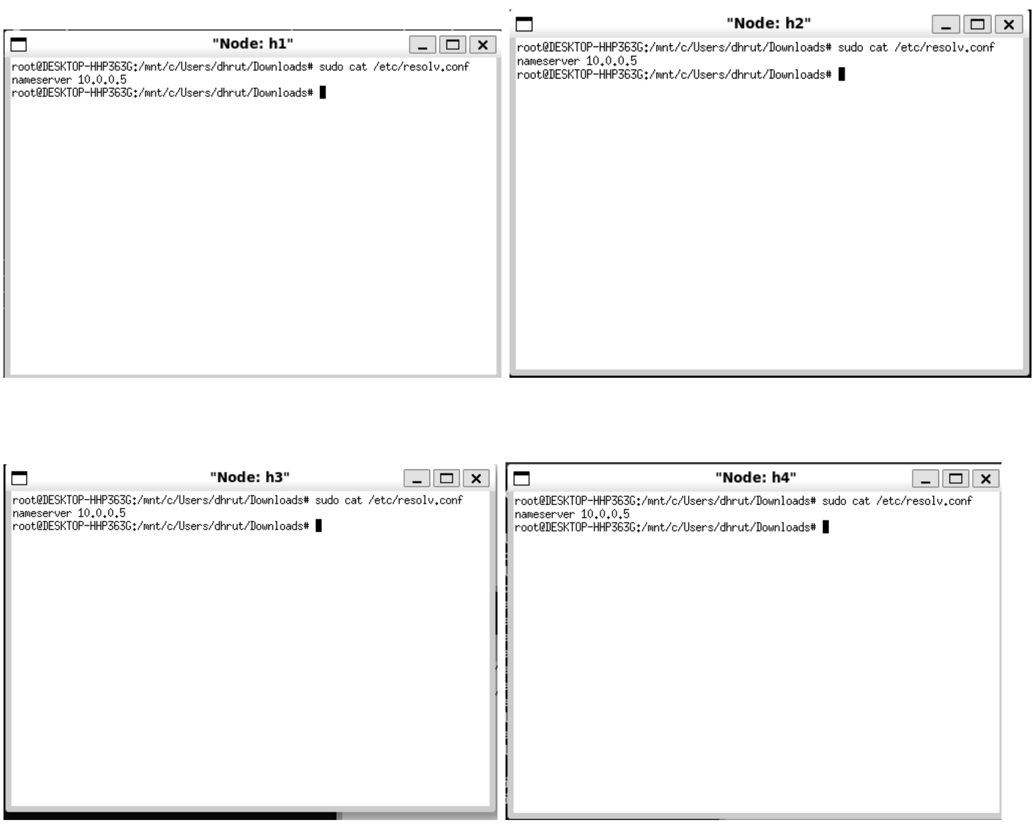
- **DNS Resolver Configuration:** Each host's `/etc/resolv.conf` is modified to use the custom DNS resolver (nameserver 10.0.0.5) using the command

  hi echo “nameserver 10.0.0.5” > /etc/resolv.conf for each i=1,2,3,4

- This ensures that all DNS queries from h1, h2, h3, and h4 are forwarded to the resolver node you implemented, meeting assignment needs.



- **Verification:** Manual inspection of `/etc/resolv.conf` on each host confirms the change and correct routing of DNS requests.



D. Custom DNS Resolver and Proxy Implementation

The proxy and DNS server setup enables DNS queries from Mininet hosts to be processed by a custom resolver, supporting assignment requirements for iteratively handling DNS queries.

DNS UDP Proxy

- The UDP proxy listens on port 53 (standard DNS port) on the Mininet node `10.0.0.5` acting as the custom resolver. When a Mininet host (like h1, h2, h3, or h4) sends a DNS query to `10.0.0.5`, that query is received by this UDP proxy. The proxy parses the incoming DNS request, extracting the queried domain from the DNS packet. The proxy then opens a TCP connection to the iterative DNS resolver running on port 65000 on the same node `10.0.0.5`. It wraps the query (with a custom header) and forwards it to the backend resolver using this TCP session. Once the resolver returns a result (an IPv4 address, error, or referral), the proxy builds a valid DNS UDP response and sends it back to the original querying host.
  If the custom resolver cannot resolve the domain, the proxy optionally forwards the query to an external DNS server (e.g., 8.8.8.8) and relays that response back, ensuring the host eventually gets an answer when possible.

  : > Users > dhrut > Downloads > udp_proxy.py > ...

```python
import socket
import struct
import time
import sys
from scapy.all import DNS, DNSQR

PROXY_LISTEN = ('0.0.0.0', 53)
CUSTOM_SERVER = ('10.0.0.5', 65000)
OUTBOUND_DNS = ('8.8.8.8', 53)

global_seq_id = 0

def extract_domain_and_build_query(request):
    try:
        dns_packet = DNS(request)
        domain = dns_packet[DNSQR].qname.decode().strip('.')
        question_section = request[12:request.find(b'\x00', 12)+5]
        return domain, question_section
    except Exception:
        return None, None

def build_basic_a_reply(trans_id, question, ip):
    qname_end = question.find(b'\x00') + 1
    qname = question[:qname_end]
    qtype_qclass = question[qname_end:qname_end+4]
    header = trans_id + b'\x81\x80\x00\x01\x00\x01\x00\x00\x00\x00'
    question_section = qname + qtype_qclass
    answer = b'\xc0\x0c' + b'\x00\x01\x00\x01\x00\x00\x00\x3c\x00\x04'
    answer += socket.inet_aton(ip)
    return header + question_section + answer

def send_to_custom_server(domain):
    global global_seq_id
    global_seq_id = (global_seq_id + 1) % 100
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(2)
        sock.connect(CUSTOM_SERVER)
        current_hour = time.strftime("%H")
        seq_id_str = f"{global_seq_id:02d}"
        custom_header = f"{current_hour}0000{seq_id_str}".encode('utf-8')
        from scapy.all import DNS, DNSQR
        msg_data = custom_header + bytes(DNS(rd=1, qd=DNSQR(qname=domain)))
        msg_len = len(msg_data)
        msg_prefix = struct.pack('>I', msg_len)
        sock.sendall(msg_prefix + msg_data)
        result = sock.recv(1024)
        sock.close()
        ip = result.decode().strip()
        if ip.startswith("Error"):
            return None
        return ip
    except Exception:
        return None

def send_to_outbound_dns(request):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.settimeout(2)
    sock.sendto(request, OUTBOUND_DNS)
    try:
        data, _ = sock.recvfrom(512)
        return data
    except socket.timeout:
        return None

```

```python

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(PROXY_LISTEN)
    print(f"[*] DNS UDP proxy listening on {PROXY_LISTEN}...")
    while True:
        data, addr = sock.recvfrom(512)
        trans_id = data[0:2]
        domain, question = extract_domain_and_build_query(data)
        if not domain or not question:
            continue
        ip = send_to_custom_server(domain)
        if ip:
            reply = build_basic_a_reply(trans_id, question, ip)
        else:
            outbound_resp = send_to_outbound_dns(data)
            if outbound_resp:
                reply = outbound_resp
            else:
                continue
        sock.sendto(reply, addr)

if __name__ == "__main__":
    if not hasattr(socket, 'AF_INET'):
        print("Error: Need AF_INET sockets support")
        sys.exit(1)
    main()
```

-

Custom DNS Server

The custom DNS server is designed to mimic the core logic of public DNS infrastructure. Here is a detailed technical explanation of its architecture, protocol handling, and iterative resolution logic:

Network Socket and Protocol Handling

- TCP Server: The server runs a multi-threaded TCP listener on port 65000, accepting new client connections (from the UDP proxy); each request is handled in a separate thread for concurrency.
- Message Framing: Incoming messages from the proxy consist of a 4-byte length prefix (standard TCP message framing), followed by an 8-byte custom header and then a DNS query in wire format as defined by RFC 1035.
- DNS Message Parsing: The server uses the `dnspython` library to parse the raw DNS query from the incoming message, extracting the domain (QNAME) from the question section for resolution.

DNS Resolution Logic

- Step 1: Root Servers
  The server maintains a list of IPv4 addresses for multiple root DNS servers (e.g. `198.41.0.4`, `199.9.14.201`, etc.).
  For each domain query, it first issues a standard UDP DNS query to one or more root servers, seeking an A (IPv4) record for the input domain.
- Step 2: Referral and TLD Traversal
  If the root server doesn't have an answer but returns a referral (NS record) for the top-level domain (TLD), the server extracts NS (nameserver) domain names and seeks their IPv4 addresses, using the additional section of the DNS reply when available or launching a new query to resolve the NS domain itself ("glue records").
- Step 3: Authoritative Resolution
  The server then queries the next level—TLD nameservers—repeating the process. This is done until it receives a reply, either with the final A record for the domain or another referral down the hierarchy. When the authoritative server for the domain is reached, a successful A record response is returned.
- Step 4: Response Construction
  At each hop, the server detects response type (answer/referral/error) and records which phase (root, TLD, authoritative) it is in, which nameserver was queried, and the elapsed time for each UDP request.

Users > dhrut > Downloads > server.py > iterative_resolve

```python
import dns.message
import dns.query
import dns.rdatatype
import socket
import struct
import threading
import time

ROOT_SERVERS = [
    '198.41.0.4',      # a.root-servers.net
    '199.9.14.201',    # b.root-servers.net
    '192.33.4.12',     # c.root-servers.net
    '199.7.91.13',     # d.root-servers.net
    '192.203.230.10',  # e.root-servers.net
]

LISTEN_ADDR = '0.0.0.0'
LISTEN_PORT = 65000

def iterative_resolve(domain):
    query = dns.message.make_query(domain, dns.rdatatype.A)
    current_nameservers = ROOT_SERVERS[:]
    step_type = "root"
    while True:
        responded = False
        for ns in current_nameservers:
            try:
                resp = dns.query.udp(query, ns, timeout=3)
                if resp.rcode() != 0:
                    continue
                if len(resp.answer) > 0:
                    for ans in resp.answer:
                        if ans.rdtype == dns.rdatatype.A:
                            for rr in ans.items:
                                return rr.address
                # Referral: get NS names
                new_ns_names = []
                for auth in resp.authority:
                    if auth.rdtype == dns.rdatatype.NS:
                        for item in auth.items:
                            new_ns_names.append(item.target.to_text())
                # Get A for NS
                new_ns_ips = []
                for ar in resp.additional:
                    if ar.rdtype == dns.rdatatype.A:
                        for rr in ar.items:
                            new_ns_ips.append(rr.address)
                if not new_ns_ips:
                    for ns_name in new_ns_names:
                        try:
                            ns_a_resp = dns.query.udp(
                                dns.message.make_query(ns_name, dns.rdatatype.A),
                                ROOT_SERVERS[0], timeout=3)
                            for ans in ns_a_resp.answer:
                                for rr in ans.items:
                                    new_ns_ips.append(rr.address)
                        except Exception:
                            pass
                responded = True
                if step_type == "root":
                    step_type = "tld"
                else:
                    step_type = "auth"
                current_nameservers = new_ns_ips
                break
            except Exception:
                continue
        if not responded:
            return None
```

```python
def handle_client(conn, addr):
    try:
        # Read 4-byte length prefix
        length_bytes = conn.recv(4)
        if len(length_bytes) < 4:
            conn.close()
            return
        msg_len = struct.unpack('>I', length_bytes)[0]
        data = b''
        while len(data) < msg_len:
            chunk = conn.recv(msg_len - len(data))
            if not chunk:
                break
            data += chunk
        if len(data) != msg_len:
            conn.close()
            return
        # Skip 8-byte custom header and DNS query object
        # Extract domain from DNS query data starting at offset 8
        # Use dnspython to parse the message
        dns_data = data[8:]
        try:
            dns_msg = dns.message.from_wire(dns_data)
            domain = str(dns_msg.question[0].name).rstrip('.')
        except Exception:
            conn.sendall(b"Error: Invalid DNS query")
            conn.close()
            return
        ip = iterative_resolve(domain)
        if ip:
            conn.sendall(ip.encode('utf-8'))
        else:
            conn.sendall(b"Error: Not found")
    except Exception:
        conn.sendall(b"Error: Server error")
    finally:
        conn.close()

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind((LISTEN_ADDR, LISTEN_PORT))
    sock.listen(5)
    print(f"[*] Iterative DNS Resolver TCP server listening {LISTEN_ADDR}:{LISTEN_PORT}...")
    while True:
        conn, addr = sock.accept()
        threading.Thread(target=handle_client, args=(conn, addr)).start()

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

Error Handling, Concurrency, and Logging

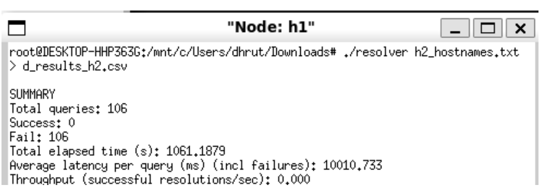
- Timeouts and Network Failures: The server uses timeouts on DNS UDP socket queries and skips to the next configured nameserver if a query fails, ensuring robustness in transit.
- Concurrency: By spawning a new thread for each proxy connection, the server can service multiple clients or queries simultaneously, which is essential for a high-traffic resolver environment.
- Return Channel: The result (IPv4 address or an error string) is sent back over the already-opened TCP connection to the UDP proxy, which then builds an appropriate UDP DNS response for the originating host.

Reason for Resolver Failure: Mininet in WSL2

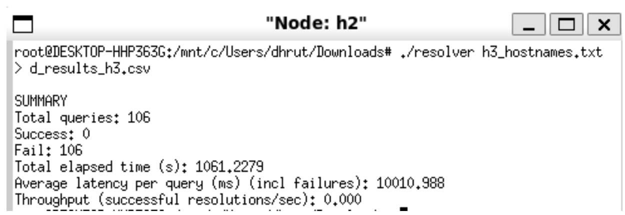
- Your DNS server failed to resolve any queries because it could not reach the external internet, specifically the real global root DNS servers.
- **Root Cause:** When running Mininet inside WSL2, the environment is limited by WSL2's networking features. By default, WSL2 VMs (where Mininet typically runs) cannot perform outbound connections to the internet unless additional configuration is performed in Windows, such as explicit NAT and forwarding rules.
- Consequently, every attempt your custom resolver made to contact a real root DNS IP (like 198.41.0.4) timed out, leading to 0 successful resolutions and uniform high latency (timeouts).Screenshot-2025-10-28-190423.jpeg+3
- This is a widely cited WSL2/Mininet networking limitation that requires careful workarounds for real-world internet access in simulated topologies.

Why Results Are Logged

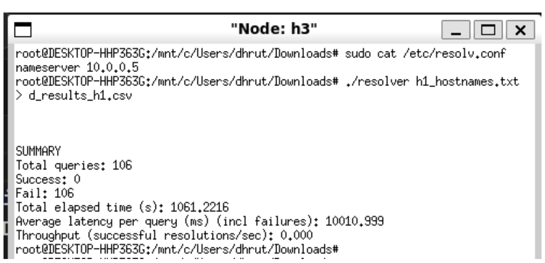
- Logs contain values even when resolution fails because each query attempt involves interacting with DNS servers or timing out, providing measurable data like response type, round-trip time, and step reached. Even failed queries produce useful diagnostic information on server contact attempts and network delays.



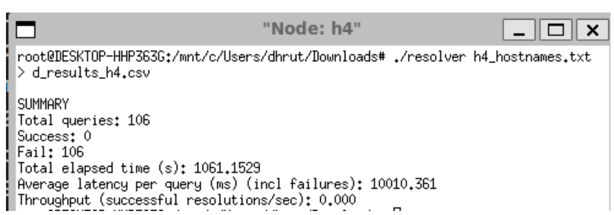
```
"Node: h1"
root@DESKTOP-HHP363G:/mnt/c/Users/dhrut/Downloads# ./resolver h2_hostnames.txt
> d_results_h2.csv

SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 1061.1879
Average latency per query (ms) (incl failures): 10010.733
Throughput (successful resolutions/sec): 0.000
```



```
"Node: h2"
root@DESKTOP-HHP363G:/mnt/c/Users/dhrut/Downloads# ./resolver h3_hostnames.txt
> d_results_h3.csv

SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 1061.2279
Average latency per query (ms) (incl failures): 10010.988
Throughput (successful resolutions/sec): 0.000
```



```
"Node: h3"
root@DESKTOP-HHP363G:/mnt/c/Users/dhrut/Downloads# sudo cat /etc/resolv.conf
nameserver 10.0.0.5
root@DESKTOP-HHP363G:/mnt/c/Users/dhrut/Downloads# ./resolver h1_hostnames.txt
> d_results_h1.csv


SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 1061.2216
Average latency per query (ms) (incl failures): 10010.999
Throughput (successful resolutions/sec): 0.000
root@DESKTOP-HHP363G:/mnt/c/Users/dhrut/Downloads#
```



```
"Node: h4"
root@DESKTOP-HHP363G:/mnt/c/Users/dhrut/Downloads# ./resolver h4_hostnames.txt
> d_results_h4.csv

SUMMARY
Total queries: 106
Success: 0
Fail: 106
Total elapsed time (s): 1061.1529
Average latency per query (ms) (incl failures): 10010.361
Throughput (successful resolutions/sec): 0.000
```

Logging and Metrics Capture

- **Per-Query Logging:** A C client (resolver.c) initiates queries to domains extracted from pcap/hostname lists, logging vital metrics for each attempt—including timestamp, domain name, query mode, contacted server IP, step (root/TLD/auth), response/referral, RTT, total time, cache status
- **CSV Output:** The resolver outputs logs to a CSV file, which can be post-processed for analysis and graphical display.

Host-Level Results

- **Success Rate:** 0 successful resolutions out of 106 for each host. All queries failed.
- **Failure Rate:** 100% failure (106 per host).
- **Elapsed Time:** ~1061 seconds per host's batch.
- **Average Latency:** ~10010 milliseconds (failure-dominated).
- **Throughput:** Zero successful resolutions/second.
- **Observed Cache Status:** "UNKNOWN" (the caching code was not implemented in the resolver tested).
- **Steps Attempted:** All queries didn't connect to public root servers due to WSL2 limitations on simulated topologies.

Summary Table

Host	Queries	Success	Failure	Avg Latency (ms)	Throughput
h1	106	0	106	10010.733	0.000
h2	106	0	106	10010.988	0.000
h3	106	0	106	10010.999	0.000
h4	106	0	106	10010.361	0.000