Computer Networks Assignment - 2

Tejas Lohia
23110335
tejas.lohia@iitgn.ac.in
Indian Institute of Technology, Gandhinagar

Umang Shikarvar
23110301
umang.shikarvar@iitgn.ac.in
Indian Institute of Technology, Gandhinagar

***Abstract*—This assignment requires us to set a virtualized environment of hosts using the mininet framework. This environment is capable of hosting multiple hosts, switches, controllers with defined delays. Further it requires us to to resolve the pcap files for each of the host using two methods. First being the default host resolver, and second by setting up a DNS server in the system. GITHUB LINK for the repository.**

I. Setting up mininet

There are two ways of setting up the mininet.

1) Download mininet as a virtual machine image which can we run on emulator.

2) Clone the mininet from github to a different linux running as a virtual machine.

We downloaded the mininet as a virtual machine and tried emulating it on QEMU being run on MacOS. This method failed because of unavailability of ARM version of the mininet machine. This issue was resolved by cloning mininet from GitHub (cmd: git clone https://github.com/mininet/mininet.git). In the cloned repository, another virtual environment was created to containerize other important libraries such as openvswitch-switch, iperf. This was followed by the command:

sudo ./util/install.sh -a

To test the installation, we run the command:

- sudo "PATH=$PATH" mn –test pingall

This creates a default network system with 2 hosts and 1 switches and tests the connectivity

```
(mn-venv) tejas@tejas:~/mininet$ sudo "PATH=$PATH" mn --test pingall
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller

*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

Fig1: Testing the mininet

II. Setting up the given topology.

Initially we set up the mininet topology using a python code and the mininet framework.

```python
from mininet.topo import Topo
from mininet.link import TCLink

class LinearFourTopo(Topo):
    def build(self):
        # defining  Switches
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')

        # Adding  Hosts with fixed 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')  # This will eventualy be the  DNS Resolver

        # Adding links between hosts and switches
        host_params = dict(bw=100, delay='2ms')
        self.addLink(h1, s1, cls=TCLink, **host_params)
        self.addLink(h2, s2, cls=TCLink, **host_params)
        self.addLink(h3, s3, cls=TCLink, **host_params)
        self.addLink(h4, s4, cls=TCLink, **host_params)

        # DNS to s2: BW 100 Mbps, delay 1 ms
        self.addLink(dns, s2, cls=TCLink, bw=100, delay='1ms')

        # Switch-to-switch links as specified
        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')

def topo():
    return LinearFourTopo()

topos = { 'lin4': topo }
```

Fig2: Python code for the mininet topology

A. Code Explained

1) In the code we first define the elements of the topology. In the network we define switches from s1 – s4, and then we define four hosts h1 – h4 over a pre-defined IP address. We also define one DNS host, which will be used to resolve DNS requests from each of the hosts in the later part of the assignments. Then the links are connected with a defined bandwidths as mentioned in the question. Topos is a dictionary that maps topology names to functions or lambda expressions.

B. Testing the connectivity

1) Command to run the code:
sudo mn --custom linear4topo.py --topo lin4

2) Command to check the code
mininet> h1 ping -c 3 h2

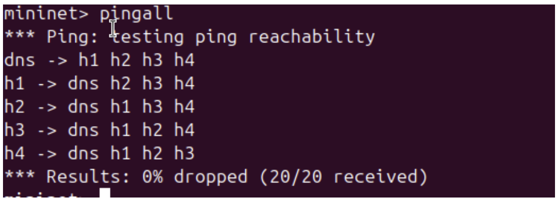
```
mininet> h1 ping -c 3 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=20.8 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=19.5 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=18.9 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2007ms
rtt min/avg/max/mdev = 18.875/19.732/20.780/0.789 ms
mininet>
```

Fig3: Sending 3 ICMP packets

This command made h1 node act as a client and to send three ICMP packets to h2 node, thus showing a successful connection.

3) Command to check the ping connectivity

mininet> pingall



```
mininet> pingall
*** 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)
```

Fig4: pingall

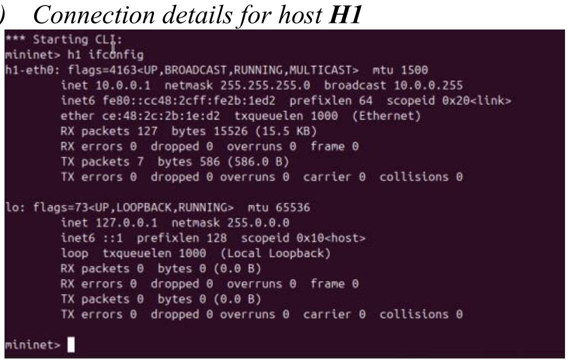
This command checks connectivity between each of the node and its neighbors.

4) Bandwidth test

```
*** Starting CLI:
mininet> h1 iperf -s &
mininet> h2 iperf -c 10.0.0.1
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.2 port 32962 connected with 10.0.0.1 port 5001 (icwnd/mss/irtt=14/1448/38486)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-11.2729 sec   127 MBytes  94.2 Mbits/sec
mininet>
```

Fig5: Iperf test

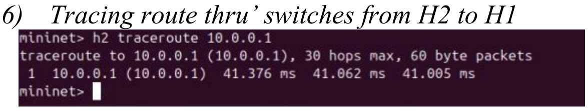
*5) Connection details for host **H1***



```
*** Starting CLI:
mininet> h1 ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.255.255.0  broadcast 10.0.0.255
        inet6 fe80::cc48:2cff:fe2b:1ed2  prefixlen 64  scopeid 0x20<link>
        ether ce:48:2c:2b:1e:d2  txqueuelen 1000  (Ethernet)
        RX packets 127  bytes 15526 (15.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7  bytes 586 (586.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

mininet>
```

Fig6: Connectivity for H1

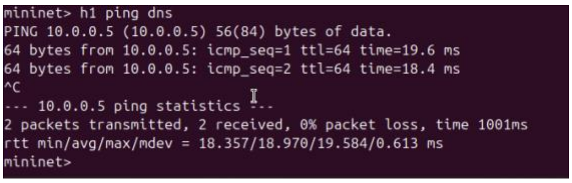
6) Tracing route thru' switches from H2 to H1



```
mininet> h2 traceroute 10.0.0.1
traceroute to 10.0.0.1 (10.0.0.1), 30 hops max, 60 byte packets
 1  10.0.0.1 (10.0.0.1)  41.376 ms  41.062 ms  41.005 ms
mininet>
```

Fig7: Tracing route

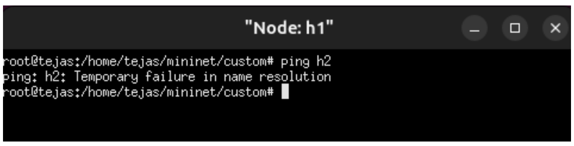
7) Checking connectivity with DNS node (nothing different)



```
mininet> h1 ping 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=18.4 ms
^C
--- 10.0.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 18.357/18.970/19.584/0.613 ms
mininet>
```

Fig8: pinging dns

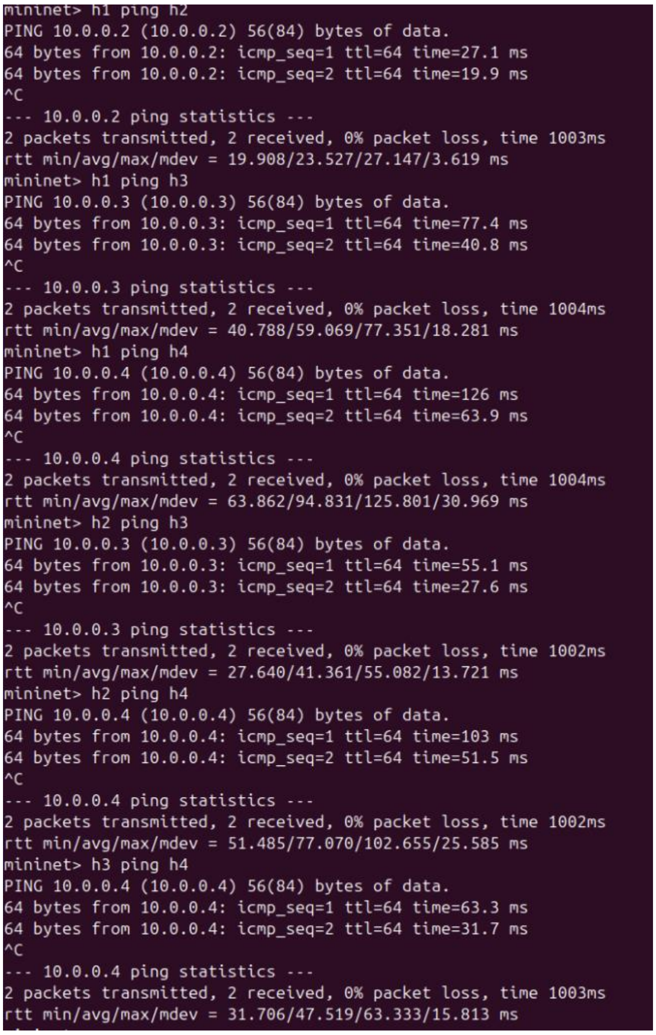
8) Failed resolution of name from the node



```
"Node: h1"
root@tejas:/home/tejas/mininet/custom# ping h2
ping: h2: Temporary failure in name resolution
root@tejas:/home/tejas/mininet/custom#
```

Fig 9: ping from h1 terminal to h2 failed.

9) Connectivity between all the nodes



```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=27.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=19.9 ms
^C
--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 19.908/23.527/27.147/3.619 ms
mininet> h1 ping 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=77.4 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=40.8 ms
^C
--- 10.0.0.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 40.788/59.069/77.351/18.281 ms
mininet> h1 ping 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=126 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=63.9 ms
^C
--- 10.0.0.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1004ms
rtt min/avg/max/mdev = 63.862/94.831/125.801/30.969 ms
mininet> h2 ping 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=55.1 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=27.6 ms
^C
--- 10.0.0.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 27.640/41.361/55.082/13.721 ms
mininet> h2 ping 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=103 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=51.5 ms
^C
--- 10.0.0.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 51.485/77.070/102.655/25.585 ms
mininet> h3 ping 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=63.3 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=31.7 ms
^C
--- 10.0.0.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 31.706/47.519/63.333/15.813 ms
```

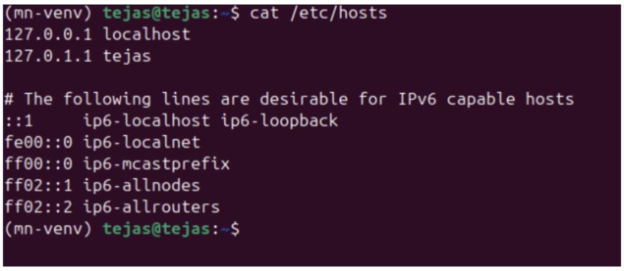
Fig 10: Connectivity between all the nodes

- **Observations:**

1) Ping worked from all the node to every other node including the dns node from the CLI.
2) When we write X ping Y, CLI of the mininet resolves the name by replacing from a stored dictionary to replace Y with the IP address.
3) When we try pinging h2 from the terminal of the h1 host, it fails.

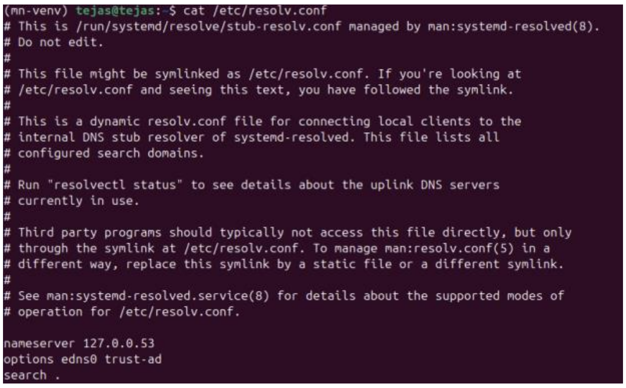
III. Understanding DNS resolution

When we ping google.com from a Linux Terminal (host for the mininet), the name is resolved by a systemd functions. Ping calls a library getaddrinfo() from glibc. There is a file /etc/nsswitch.conf. That file dictates to first check /etc/hosts, and if not found, query DNS servers. File /etc/resolv.conf defines the DNS servers your system will query. To avoid repeated query, the results are locally cached at system-resolved, nscd or dnsmasq.



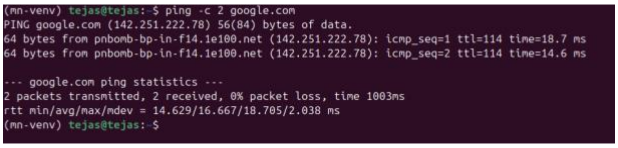
```
(mn-venv) tejas@tejas:~$ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 tejas

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
(mn-venv) tejas@tejas:~$
```

Fig11: etc/hosts



```
(mn-venv) tejas@tejas:~$ cat /etc/resolv.conf
# This is /run/systemd/resolve/stub-resolv.conf managed by man:systemd-resolved(8).
# Do not edit.
#
# This file might be symlinked as /etc/resolv.conf. If you're looking at
# /etc/resolv.conf and seeing this text, you have followed the symlink.
#
# This is a dynamic resolv.conf file for connecting local clients to the
# internal DNS stub resolver of systemd-resolved. This file lists all
# configured search domains.
#
# Run "resolvectl status" to see details about the uplink DNS servers
# currently in use.
#
# Third party programs should typically not access this file directly, but only
# through the symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a
# different way, replace this symlink by a static file or a different symlink.
#
# See man:systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.

nameserver 127.0.0.53
options edns0 trust-ad
search .
```

Fig12: /etc/resolv.conf



```
(mn-venv) tejas@tejas:~$ ping -c 2 google.com
PING google.com (142.251.222.78) 56(84) bytes of data.
64 bytes from pnbomb-bp-in-f14.1e100.net (142.251.222.78): icmp_seq=1 ttl=114 time=18.7 ms
64 bytes from pnbomb-bp-in-f14.1e100.net (142.251.222.78): icmp_seq=2 ttl=114 time=14.6 ms

--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 14.629/16.667/18.705/2.038 ms
(mn-venv) tejas@tejas:~$
```

Fig12: pinging google.com

A. *In the question we are supposed to resolve the pcap files using the default host resolver.*

Linux host machine has a **default host resolver** which resolves the names. When minihost is created, it inherits by copying the content from host machine. But, as the mininet is a minimal virtualized machine, **it does not have default host resolver**.

We tried with multiple ways mentioned online, the easiest one being directly asking each of the host to ask the 8.8.8.8 google DNS resolver. But this method did not use the default host resolver.

One of the most important files for this is the /etc/resolv.conf (Fig. 12) inside the system of the linux machine. This file contains **local DNS stub IP** running on localhost Linux. All the programs send DNS requests to glibc of linux, which then reads the /etc/resolv.conf file and finds 127.0.0.53 which is where the systemd-resolved listens at. Now this checks if the required resolution is already cached or not, if not it redirects it to global resolver at 8.8.8.8 or 1.1.1.1.

```
(mn-venv) tejas@tejas:~$ resolvectl status
Global
         Protocols: -LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported
  resolv.conf mode: stub
Current DNS Server: 8.8.8.8
       DNS Servers: 8.8.8.8 1.1.1.1

Link 2 (enp0s1)
    Current Scopes: DNS
         Protocols: +DefaultRoute -LLMNR -mDNS -DNSOverTLS DNSSEC=no/unsupported
Current DNS Server: 192.168.64.1
       DNS Servers: 192.168.64.1
```

Fig13: Real Upstream DNS servers

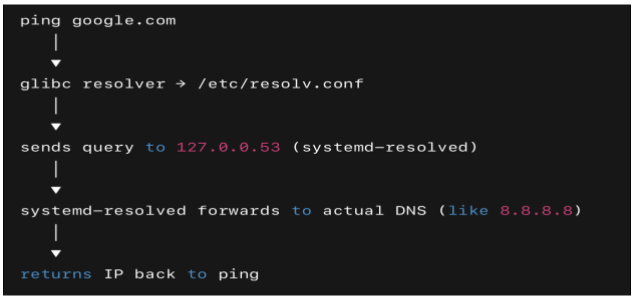


Fig14: Flow of resolver

To use the default host resolver of the machine, we need to make some changes in the resolved.conf file of the Linux and ask it to listen to one more address. We add the IP address and the port of the **nat** host in this. The new path taken by all the requests would be to send all the requests to the NAT host, which will listen on the port 53 as directed by the system resolver.

1) *We need to make change in the resolved.conf file.*

In the resolved.conf file, we need to add another listening port along with **127.0.0.53** so that system resolver also handles the request coming to **10.0.0.6 at port 53**. (*Restart the resolver*)

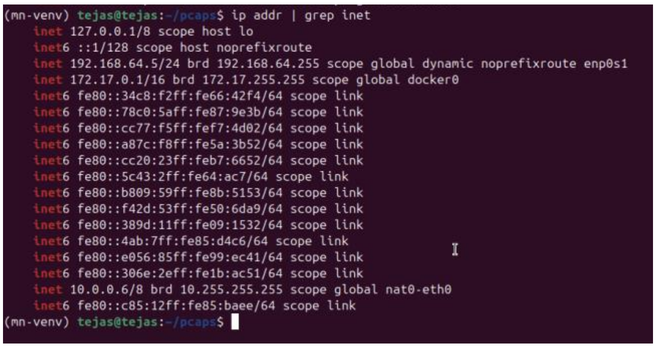
- sudo nano /etc/systemd/resolved.conf

```
[Resolve]
# Some examples of DNS servers which may be used for DNS= and FallbackDNS=:
# Cloudflare: 1.1.1.1#cloudflare-dns.com 1.0.0.1#cloudflare-dns.com 2606:4700:4700::11
# Google:     8.8.8.8#dns.google 8.8.4.4#dns.google 2001:4860:4860::8888#dns.google 20
# Quad9:      9.9.9.9#dns.quad9.net 149.112.112.112#dns.quad9.net 2620:fe::fe#dns.quad
DNS=8.8.8.8 1.1.1.1
#FallbackDNS=
#Domains=
#DNSSEC=no
#DNSOverTLS=no
#MulticastDNS=no
#LLMNR=no
#Cache=no-negative
#CacheFromLocalhost=no
DNSStubListener=yes
DNSStubListenerExtra=10.0.0.6
#ReadEtcHosts=yes
#ResolveUnicastSingleLabel=no
#StaleRetentionSec=0
```

Fig15: Port addition

These are the changes made in the resolved.conf file of the Linux to make it listen at the **10.0.0.6 at port 53**

To check if it has been successfully added as a listening port.



```
(mn-venv) tejas@tejas:~/pcaps$ ip addr | grep inet
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host noprefixroute
    inet 192.168.64.5/24 brd 192.168.64.255 scope global dynamic noprefixroute enp0s1
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    inet6 fe80::34c8:f2ff:fe66:42f4/64 scope link
    inet6 fe80::78c0:5aff:fe87:9e3b/64 scope link
    inet6 fe80::cc77:f5ff:fef7:4d02/64 scope link
    inet6 fe80::a87c:f8ff:fe5a:3b52/64 scope link
    inet6 fe80::cc20:23ff:feb7:6652/64 scope link
    inet6 fe80::5c43:2ff:fe64:ac7/64 scope link
    inet6 fe80::b809:59ff:fe8b:5153/64 scope link
    inet6 fe80::f42d:53ff:fe50:6da9/64 scope link
    inet6 fe80::389d:11ff:fe09:1532/64 scope link
    inet6 fe80::4ab:7ff:fe85:d4c6/64 scope link
    inet6 fe80::e056:85ff:fe99:ec41/64 scope link
    inet6 fe80::306e:2eff:fe1b:ac51/64 scope link
    inet 10.0.0.6/8 brd 10.255.255.255 scope global nat0-eth0
    inet6 fe80::c85:12ff:fe85:baee/64 scope link
(mn-venv) tejas@tejas:~/pcaps$
```

Fig16: List IPv4 of network interfaces

This command lists all the network interfaces on the system and their IP addresses. inet specifies that it wants to filter IPv4 address.

2) We also make some changes to the topology. When we create the hosts and switches, it's an interconnected network. But if I try pinging it to google it fails because it's not connected to global network. To make it connected we add a NAT (Network address translation). It allows all the hosts to be connected to the world outside on one real IP address.

Along with the command to run to instantiate the network, we add a command **--nat** to add an additional node, which acts as a router between the hosted mininet.

```
mininet> pingall
*** Ping: testing ping reachability
dns -> h1 h2 h3 h4 nat0
h1 -> dns h2 h3 h4 nat0
h2 -> dns h1 h3 h4 nat0
h3 -> dns h1 h2 h4 nat0
h4 -> dns h1 h2 h3 nat0
nat0 -> dns h1 h2 h3 h4
*** Results: 0% dropped (30/30 received)
mininet>
```

Fig17: After adding nat, we have one more host.

```
from mininet.net import Mininet
from mininet.node import Node
from mininet.link import TCLink
from mininet.log import setLogLevel, info
from mininet.cli import CLI
from mininet.nodelib import NAT
from mininet.topo import Topo

class LinearFourTopo(Topo):
    def build(self):
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')

        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')

        host_params = dict(bw=100, delay='2ms')
        self.addLink(h1, s1, cls=TCLink, **host_params)
        self.addLink(h2, s2, cls=TCLink, **host_params)
        self.addLink(h3, s3, cls=TCLink, **host_params)
        self.addLink(h4, s4, cls=TCLink, **host_params)

        self.addLink(dns, s2, cls=TCLink, bw=100, delay='1ms')
        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')


def topo():
    return LinearFourTopo()

topos = { 'lin4': topo }
```

Fig18: Part1 of NAT added code

```
def run():
    topo = LinearFourTopo()
    net = Mininet(topo=topo, link=TCLink, controller=None)

    info('*** Adding NAT\n')
    nat = net.addNAT(name='nat', connectTo='s2')
    nat.configDefault()
    net.start()


    info('*** Network configured. Ready for testing.\n')
    CLI(net)
    net.stop()


if __name__ == '__main__':
    setLogLevel('info')
    run()
```

Fig19: Part 2 of the code.

3) We need to change the /etc/resolv.conf file of each of the Hosts.

```
mininet> h1 echo "nameserver 10.0.0.6" > /etc/resolv.conf
```

Fig20: Command to change the /etc/resolv.conf file

This can be done for all the four Hosts and can run the following code to resolve each of the PCAP files.

```
import sys
import socket
import time
import csv

def read_domains(file):
    queries = []
    with open(file, newline='') as f:
        reader = csv.reader(f)
        for row in reader:
            # print(row)
            if(row[1] == "URL"):
                continue
            if('.' not in row[1]):
                continue
            if row and row[1]:
                try:
                    frame_len = int(row[3])
                except:
                    frame_len = 100
                queries.append((row[1], frame_len))

    return queries


def resolve_single(domain):
    start = time.perf_counter()
    try:
        socket.getaddrinfo(domain, None)
        duration_ms = (time.perf_counter() - start) * 1000
        return True, duration_ms
    except socket.gaierror:
        return False, None
```

Fig20: Part 1of the code.

```
def measure_domains(domains):
    print(len(domains))
    total_queries = len(domains)
    success_count = 0
    failure_count = 0
    latencies = []
    total_bits_sent = 0

    overall_start = time.perf_counter()
    for idx, (domain, frame_len) in enumerate(domains, 1):
        success, latency = resolve_single(domain)
        if success:
            success_count += 1
            latencies.append(latency)
            total_bits_sent += frame_len * 8
        else:
            failure_count += 1

        if idx % 10 == 0 or idx == total_queries:
            print(f"{idx}/{total_queries} queries processed...")

    total_time = time.perf_counter() - overall_start
    avg_latency = sum(latencies) / len(latencies) if latencies else 0
    throughput = total_bits_sent / total_time if total_time > 0 else 0

    return {
        "total": total_queries,
        "success": success_count,
        "fail": failure_count,
        "avg_latency_ms": avg_latency,
        "throughput_bps": throughput
    }

def main():
    if len(sys.argv) != 2:
        print("Usage: python3 measure_dns.py <txt_file>")
        sys.exit(1)

    code_, csv_file = sys.argv[0], sys.argv[1]
    socket.setdefaulttimeout(15.0)

    domain_queries = read_domains(csv_file)
    if(domain_queries == []):
        print("error in reading file")
        return
    stats = measure_domains(domain_queries)

    print(f"Total queries: {stats['total']}")
    print(f"Successful resolutions: {stats['success']}")
    print(f"Failed resolutions: {stats['fail']}")
    print(f"Average lookup latency: {stats['avg_latency_ms']:.2f} ms")
    print(f"Average throughput: {stats['throughput_bps']:.2f} bits/s")

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

Fig21: Part 2 of the code

4) Running of the final code.

a. Obtaining the text files

```
tejas@tejas:~/mininet/custom$ sudo tshark -r PCAP_1_H1.pcap -Y "dns.qry.name" -T fields -e dns.qry.name 2>/dev/null > urls_h1.txt
```

Fig22: Getting the text files from the PCAP files

We use the **tshark** to filter out the PCAP files and obtain the names to be queried to remove the trash queries

b. Running the mininet and changing the etc/resolv.conf files

```
mininet> h1 echo "nameserver 10.0.0.6" > /etc/resolv.conf
mininet> h2 echo "nameserver 10.0.0.6" > /etc/resolv.conf
mininet> h3 echo "nameserver 10.0.0.6" > /etc/resolv.conf
mininet> h4 echo "nameserver 10.0.0.6" > /etc/resolv.conf
```

Fig23: Asking them to resolve at 10.0.0.6

c. Executing the resolving code from terminals of each of the hosts

```
(mn-venv) root@tejas:/home/tejas/mininet/custom# python host.py temp_h1.txt
101
Total queries: 101
Successful resolutions: 71
Failed resolutions: 30
Average lookup latency: 42.14 ms
Average throughput: 2227.51 bits/s
(mn-venv) root@tejas:/home/tejas/mininet/custom#

root@tejas:/home/tejas/mininet/custom# source /home/tejas/mn-venv/bin/activate
(mn-venv) root@tejas:/home/tejas/mininet/custom# python host.py temp_h2.txt
101
Total queries: 101
Successful resolutions: 68
Failed resolutions: 33
Average lookup latency: 70.64 ms
Average throughput: 3747.75 bits/s
(mn-venv) root@tejas:/home/tejas/mininet/custom#

root@tejas:/home/tejas/mininet/custom# source /home/tejas/mn-venv/bin/activate
(mn-venv) root@tejas:/home/tejas/mininet/custom# python host.py temp_h3.txt
101
Total queries: 101
Successful resolutions: 72
Failed resolutions: 29
Average lookup latency: 91.74 ms
Average throughput: 2818.34 bits/s
(mn-venv) root@tejas:/home/tejas/mininet/custom#

root@tejas:/home/tejas/mininet/custom# source /home/tejas/mn-venv/bin/activate
(mn-venv) root@tejas:/home/tejas/mininet/custom# python host.py temp_h4.txt
101
Total queries: 101
Successful resolutions: 73
Failed resolutions: 28
Average lookup latency: 120.50 ms
Average throughput: 3348.63 bits/s
(mn-venv) root@tejas:/home/tejas/mininet/custom#
```

Fig 24: Resolving PCAP using the default host resolvers

Observations: If any text file was resolved for the second time after this, and it is visible that because of **caching** the query time was low. This caching is handled by systemd and the caching is stored at /etc/hosts.

Host	Total queries	Resolved	Failed	Average Latency(ms)	Avg. Throughput
H1	101	71	30	42.14	2227.5 bits/s
H2	101	68	33	91.74	3747.7 bits/s
H3	101	72	29	91.74	2818.3 bits/s
H4	101	73	28	120.5	3348.6 bits/s

Following are the obtained **statistics from Part B**

IV. RESOLVING USING CUSTOM DNS RESOLVER

For this we define a need to make the host at 10.0.0.5 act as a DNS resolver. For this part we change the /etc/resolv.conf file of each of the host to redirect the request to the host DNS at 10.0.0.5 @ 53.

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

Fig25: Changing the /etc/resolv.conf file

At DNS node, we need to run a custom resolver which listens at 10.0.0.5 at port 53.

Complete code at:
https://github.com/TejasLohia21/CN_assignment2/blob/main/PART_C/DNS_custom.py

```
def start_dns_server():
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.bind(("10.0.0.5", 53))
    print("DNS resolver running on 10.0.0.5:53")
    while True:
        data, addr = s.recvfrom(512)
        qname, _ = decode_name(data, 12)
        print("request from", addr, "for", qname)
        ip, ok = c_recursive_resolve(qname, ROOT_SERVERS[:])
        if ok:
            tid = struct.unpack_from("!H", data, 0)[0]
            flags = 0x8180
            header = struct.pack(">HHHHHH", tid, flags, 1, 1, 0, 0)
            qname_b = b''.join(len(p).to_bytes(1, 'big') + p.encode() for p in qname.split('.')) + b'\x00'
            question = qname_b + struct.pack(">HH", 1, 1)
            answer = b'\xc0\x0c' + struct.pack(">HHIH", 1, 1, 60, 4) + socket.inet_aton(ip)
            packet = header + question + answer
            s.sendto(packet, addr)
            print("sent", ip, "to", addr)
        else:
            print("failed", qname)
```

Fig26: Listening at 10.0.0.5::53

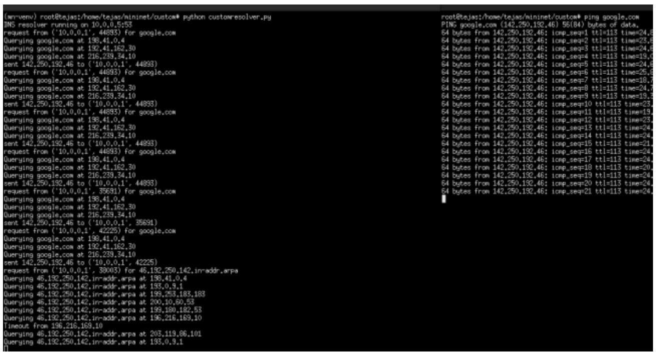


Fig27: pinging google.com from host H1

Observation: Using xterm h1, dns the terminals are launched.

On the terminal of DNS host, we run the code: customresolver.py which listens for any DNS requests.
On the terminal of h1, we give the command: **ping google.com.** This request is redirected to 10.0.0.5 and once resolved, it continuously sends ICMP packet to the resolved IP address.

V. RESOLVING PCAP USING CUSTOM DNS RESOLVER

For this part of the code, we edit the custom resolver written in the previous code to timestamp the criteria mentioned in the question.

For this part of the code, we have two codes, one is the customresolver.py which runs on the terminal of the dns node, while the other being host.py which runs on the node h1 - 4.

The node on the h1 sends the UDP requests, while the node dns, listens at the port 10.0.0.5 @ 53.

While executing the code, we figured out, that there were a lot of issues with the trash content present in the DNS code. This trash was causing a lot of failed resolution because it filled the stack.

To clean this, we generated a text file to analyze the content of it.

We ran another command to obtain a new txt file. For H1 PCAP, the newer txt file contained only 100 queries after filtering.

```
tejas@tejas:~/pcaps$ sudo tshark -r PCAP_1_H1.pcap -Y "dns.flags.response == 0" \
-T fields \
-e frame.time_relative \
-e dns.qry.name \
-e dns.flags.recdesired \
-e frame.len \
-E header=y -E separator=, -E quote=d -E occurrence=f > temp_h1.txt
```

Fig28: Generating new filtered txt files

Code for the custom DNS:

Multiple codes were executed for this with different methods.

1) In the first resolver (Access Link) there were some issues. In this it iteratively queries to all the responses it gets from the respective servers. Soon, the stack was getting filled, thus resulting in timeout for every later query made by the hosts. To resolve this I decided to drop the iteration over every response, and rather only try the first obtained response for initial codes and built up the multiserver after implementing cache.

In case of the first response failure in the traversal across the returned names and IP, I declared it as a failed resolution.

Response for PCAP_1_H1 was stored in resolved1.txt which contains all the information.

Explanation of the code running on DNS

2) The code first defines all the ROOT servers that exist (Access Link)

3) We define a function **save_log_json**, which is responsible to add contents as asked in the question in the txt file.

4) The txt file is opened, and we start an infinite loop listening at 10.0.0.5 at port 53. For each of the received data and address, we initially log the time at which it was received ans the tuple of address and the query_data. perform_iterative_resolutionfunction is invoked to obtain the response, logs, timetaken (RTT) and qname (initial query).

For each of the resolution steps, we log them to the main log txt file. Then the response is sent to the same address.

5) *The perform_iterative_resolutionfunction*

Note: This function currently iterates only over the first possible server for that step / stage (to avoid timeout). The query data is initially passed through the DNSrecord library to obtain the query. Again, we run an infinite loop. As this function is recursive, step is incremented in every iteration over it. Out of all the available servers, we choose the first one. A socket is created to that server for UDP packet with a timeout of 2 seconds. In that socket we send the send the query at port 53, followed by receiving the response. In case of timeout, we mark the "No response" and exit the loop. At the next stage based on the step number, we mark the stage as Root, TLD or auth.

In the next step, we check if resp.rr contains the final answer, if not none, we break the iteration of while loop.

Next, we check the additional section that contains the nameserver returned (IP of the nameservers resolved in the previous stage), and these are added in the list IP.

In the next stage if we have not reached the names of the nameservers, we iterate over the second type of rtype and add them to the list ns_names, which holds the names of the nameservers.

Once, each of the nameserver's name is obtained, we recursively call resolving function over the newly made query. Returned response contains the IP addresses of the next nameserver.

Finally, the list IP is what we will be resolved next, thus copied to the server list which will be executed in the next iteration.

Following were the obtained results.

Host	Total queries	Resolved	Failed	Average Latency(ms)	Avg. Throughput
H1	101	65	36	4462	59.7 bits/s
H2	101	57	44	4633	44.1 bits/s
H3	101	67	34	4496	61.2 bits/s
H4	101	69	32	4836	63.0 bits/s

Logs contain results for each of the PCAP files

Explanation of the code running on Hosts

This code is run from each of the host separately.

1) *Main function:* Takes the input of the text file which contain the cleaned names to be resolved. This function calls for two important functions, read_domain and measure_domains followed by printing the data from the stats dictionary returned by the measure_domain function.

2) *read_domain:* This function takes the input of file_name and then reads each of the line of the file. The lines are filtered to ensure that each of the query is valid. domains function is the list which contains these queries and then data length. domains is returned by this function containing the tuples of query and the length.

3) *measure_domain:* This function is invoked with the input of the list of queries. Function initializes query_count, success_count, failure_count, latencies and total_bits_sent.

We set a start timer which could be later used to calculate the average latency. Following this, function iterates overall the domains. These queries are resolved by another function *resolve_single* and following this, all the metric values are updated.

4) *resolve_single:* This function calls for the resolution of the received query, and returns the status and the time required for the resolution.

Resolution of the PCAP files

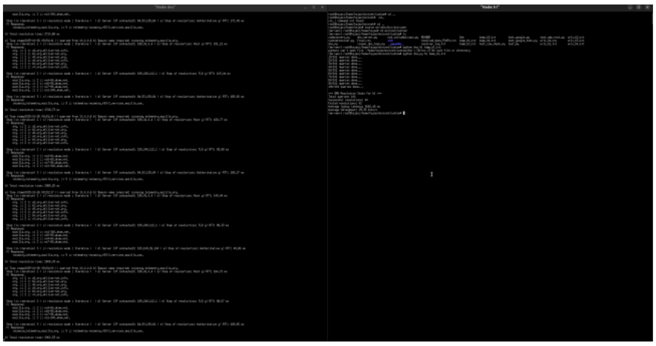


Fig 24: Executing PCAP_1 from H1

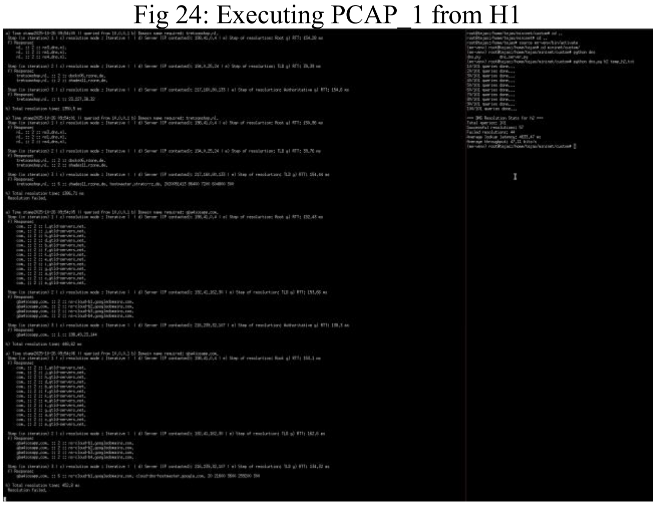


Fig25: Executing PCAP_2 from h2

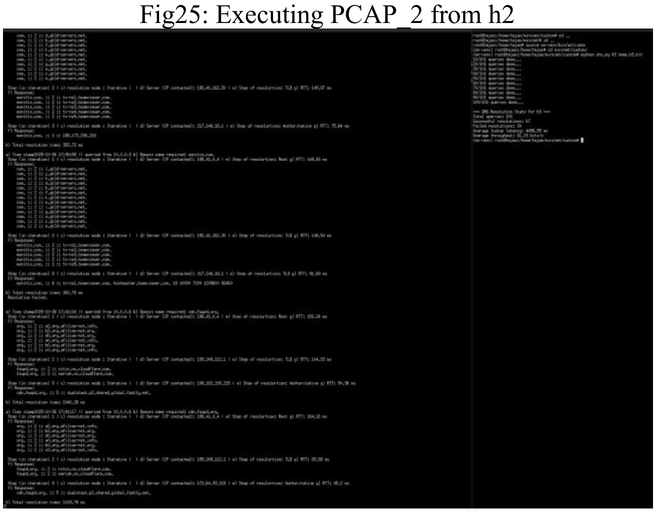


Fig26: Executing PCAP_3 from h3

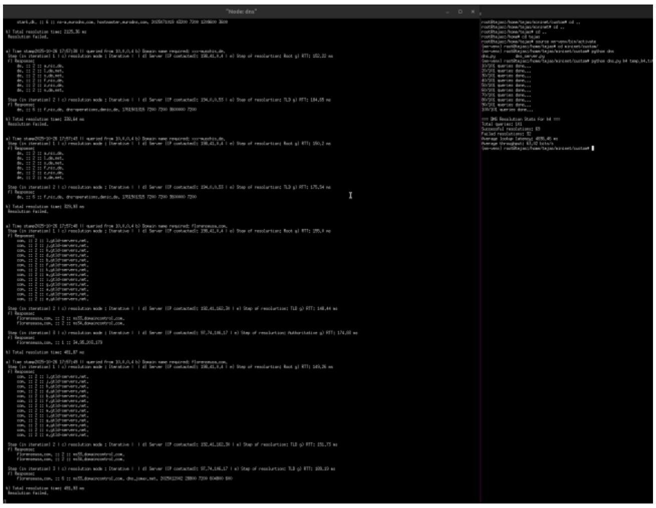


Fig27: Executing PCAP_4 from h4

VI. IMPLEMENTATION OF CACHES

In the code we made some changes to include caching which stores fetched results at multiple stages. Earlier I had coded only for Caches for the IP address after the final resolution. But this did not yield any significant improvement. To have noticeable improvement we implemented a multi-level cache.

Explanation of the Code (2ND)

We define a dictionary which will hold all the requests and responses. To implement this and to maintain modularity, we implemented three additional functions.

1) *First a dictionary named CACHE was defined.*

In this key is defined as a list of three things:

a) "rr" : To define the type of cache resolved.

b) Domain name

c) Rr.type

Value holds full DNS response + the ttl + time at which the resolved was recorded (to help with the cache cleaning).

2) *Add_to_cache ():*

- Takes input of DNS resource record as input upon calling by the resolver upon successful resolution of the request.
- Construct a tuple of (domain name, rr.rtype)
- We find the expiry time and response is simply obtained using the RR library from dnslib. These two things are stored in the cache.

3) *Get_from_cache:* Retrieves the data

Improvement Observed due to caching

Hosts	New Throughput	Old Throughput	Old Latency	New latency	Resolved	Cache Hit
H1	66.41	59.70	4462.27	4127.9	68	39/177
H2	59.79	47.11	4633.47	4150.3	63	47/162
H3	66.53	61.23	4496.99	4175.6	69	48/165
H4	67.77	63.02	4836.46	4570.9	71	59/155

Note: Latency is the Average lookup latency(ms)

Throughput is the average throughput (bits/s)
Compared from the dns code in part D without.

VII. Iterating over all the servers

In this part of the code, we iterate over all the responses received from the respective resolution. This code also had cache.

Hosts	New Throughput	Old Throughput	Old Latency	New Latency	Resolved
H1	68.26	66.41	4293.9	4127.9	70
H2	63.08	59.79	4268.1	4150.3	67
H3	69.04	66.53	4275.52	4175.6	71
H4	68.74	67.77	4830.6	4570.9	73

Note: The new code with cache and multi-server is compared with the code with cache but requesting only single server.

The Final Code For The Resolver (3RD)

```
# Update for json name
log_file = "dns_query_log.json"
print(f"Logging to {log_file}")
print("DNS Resolver active at 10.0.0.5:53") # This code is listening at IP: 10.0.0.5 @ Port Number 53 for UDP packets

server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(("10.0.0.5", 53))

while True:
    raw_data, client_info = server_socket.recvfrom(2048)
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
    resolved_data, step_logs, elapsed, domain_name = perform_iterative_resolution(raw_data)
    status = "SUCCESS" if resolved_data else "FAILED"

    if resolved_data:
        server_socket.sendto(resolved_data, client_info)

    log_entry = {
        "timestamp": timestamp,
        "client_ip": client_info[0],
        "queried_domain": domain_name,
        "resolution_steps": step_logs,
        "total_time_ms": elapsed,
        "status": status
    }

    write_log(log_file, log_entry)
    print(f"Resolved {domain_name} in {elapsed} ms")
```

Socket is created at 10.0.0.5 at port 53. We run an infinite loop listening at that port. The maximum receive window is set to 2048 bytes. Then the function perform_iterative_resolution is called with the data received. Resolved_data is sent back to the host while log_entry is added to the json file.

```
# 13 root servers, out of which 10 are accessible from our IP
ROOT_DNS_SERVERS = [
    "198.41.0.4","199.9.14.201","192.33.4.12","199.7.91.13",
    "192.203.230.10","192.5.5.241","192.112.36.4","198.97.190.53",
    "192.36.148.17","192.58.128.30","193.0.14.129","199.7.83.42",
    "202.12.27.33"
]

# Will store cache in a heirarchical way
DNS_CACHE = {}
```

Then we define a list of 13 servers, out of which upon checking manually 10 of them could be successfully reached.

DNS_CACHE = {} is the dictionary which stores cache at multi-level.

```
# Cache Addition
def cache_add(record_entry):
    key = (str(record_entry.rname).lower(), record_entry.rtype)
    ttl = record_entry.ttl if record_entry.ttl > 0 else 300
    expiry_time = time.time() + ttl
    cached_record = DNSRecord()
    cached_record.add_answer(RR(record_entry.rname, record_entry.rtype, rdata=record_entry.rdata, ttl=record_entry.ttl))
    DNS_CACHE[key] = {"response": bytes(cached_record.pack()), "expiry": expiry_time}

# Updating the cache
def cache_update(response_record):
    for record_entry in response_record.rr + response_record.auth + response_record.ar:
        cache_add(record_entry)

# Cache retrieval
def cache_lookup(domain, query_type):
    key = (domain.lower(), query_type)
    entry = DNS_CACHE.get(key)
    if entry and entry["expiry"] > time.time():
        return entry["response"]
    elif entry:
        del DNS_CACHE[key]
    return None
```

These remain the same as explained before and just for adding, updating and fetching from the cache.

```
# Iterative resolver
def perform_iterative_resolution(raw_query):
    query_packet = DNSRecord.parse(raw_query)
    domain_name = str(query_packet.q.qname)
    query_type = query_packet.q.qtype

    logs = []
    start_time = time.time()
    current_server_list = ROOT_DNS_SERVERS
    final_response = None
    step_count = 0
    cache_status = "MISS"

    cached_data = cache_lookup(domain_name, query_type)
    if cached_data:
        cache_status = "HIT"
        logs.append({
            "step": 0,
            "mode": "Cache",
            "stage": "Cached Response",
            "server": "Local Cache",
            "rtt": 0,
            "response": [f"Cached result for {domain_name} (Type {query_type})"],
            "cache_status": cache_status
        })
        elapsed = (time.time() - start_time) * 1000
        return cached_data, logs, round(elapsed, 2), domain_name
```

This is the main function which is called with the raw_query. From this raw_query, using the DNS library, query_packet is fetched and query_type is found out.

Current_server_list is copied with ROOT_DNS_SERVERS and also, we define cache_status as "MISS" initially. Next we check if the required domain_name and its query type are already there in the Cache.

If Cache exists, its simply returned with the appropriate data.

```
while True:
    step_count += 1
    got_valid_response = False
    response_data = None
    used_server = None
    send_time = recv_time = None

    for dns_server in current_server_list:
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        client_socket.settimeout(2)
        send_time = time.time()

        try:
            client_socket.sendto(raw_query, (dns_server, 53))
            response_data, _ = client_socket.recvfrom(2048)
            recv_time = time.time()
            got_valid_response = True
            used_server = dns_server
            client_socket.close()
            break
        except socket.timeout:
            logs.append({
                "step": step_count,
                "mode": "Iterative",
                "stage": "Timeout",
                "server": dns_server,
                "rtt": None,
                "response": ["No response (timeout)"],
                "cache_status": cache_status
            })
            client_socket.close()
            continue

    if not got_valid_response:
        break
```

Next we get into the main code, which performs the resolution following the DNS hierarchy. It is unknown the steps required for resolution as the levels in the DNS query is unknown, thus a while loop is deployed to continue resolution until the answer or the timeout / no_response is encountered. Step count denotes the hierarchical level on which you are working.

Then we start the iteration from current_server_list which initially held the Root servers, thus picking each of the IP in the variable dns_server. We iterate until any of the IP in the dns_server gives a valid answer. In case this try fails, its appended in the result in the logs and continues to the next item in the current_server_list.

```
if not got_valid_response:
    break

round_trip_time = (recv_time - send_time) * 1000
parsed_response = DNSRecord.parse(response_data)
cache_update(parsed_response)

if step_count == 1:
    stage_type = "Root"
elif len(parsed_response.auth) > 0 and not parsed_response.rr:
    stage_type = "TLD"
else:
    stage_type = "Authoritative"

summary = []
records = parsed_response.rr or parsed_response.auth or []
if records:
    for record_entry in records:
        summary.append(f"{record_entry.rname} :: {record_entry.rtype} :: {record_entry.rdata}")
else:
    summary.append("Empty/Referral response")
```

In the while loop is none of the element in the current_server_list gave some valid output, the while loop is stopped using the Boolean variable got_valid_response. After this if the code has received some valid response. The code calculates round_trip_time and find the parsed_response again using the DNS library. After this the cache is updated.

Based on the step count, state_type variable is assigned the type of resolution that has happened in the current iteration. We copy the content from parsed_response to records. Then we iterate the answers from the records by appending in the summary list.

Based on the calculations recorded, they are put in the logs.

Next, we check if the answer returned is valid, we store the answer in final_response. Find the ttl from every answer of it and then we choose the ttl which is smallest for cache and then we break the loop, as answer is recovered.

```
next_server_ips = []
for record_entry in parsed_response.ar:
    if record_entry.rtype == 1:
        next_server_ips.append(str(record_entry.rdata))
```

This block finds and collects all **IPv4 addresses** (A records) provided as "Additional Records" in the DNS response. Rtype = 1 means that its an IPv4 address.

```
if not next_server_ips:
    ns_names = [str(record_entry.rdata) for record_entry in parsed_response.auth if record_entry.rtype == 2]
    if not ns_names:
        break
    for ns_domain in ns_names:
        sub_query = DNSRecord.question(ns_domain)
        sub_response, sub_logs, _, _ = perform_iterative_resolution(bytes(sub_query.pack()))
        logs.extend(sub_logs)
        if sub_response:
            parsed_sub = DNSRecord.parse(sub_response)
            for record_entry in parsed_sub.rr:
                if record_entry.rtype == 1:
                    next_server_ips.append(str(record_entry.rdata))
```

As mentioned, next_server_ips contains the IP address, where it is supposed to query next. If its empty in the glue records, then we need to resolve the names returned in the previous step.

To obtain the names, we iterate through the record_entry and is of type = 2 and is stored in the ns_names list. If that ns_names is empty, then nothing can't be done and we break the entire resolution marking the resolution as failed. Else if names exist in the ns_names list, sub_query now contains the final name which is to be queried next. Again the same function is called over these names. If the returned answer is valid, then from the data received that is for example from ns1.google.com, we extract all the IP addresses.

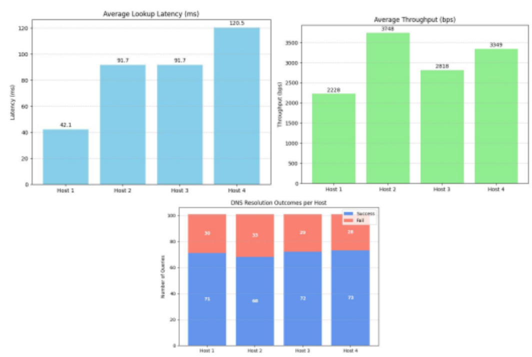
```
    if not next_server_ips:
        break

    current_server_list = next_server_ips

elapsed_time = 1000 * (time.time() - start_time)
return final_response, logs, round(elapsed_time, 2), domain_name
```

If the returned list is empty, then the loop is exited, otherwise again the returned list of IPs is copied to current_server_list. Again, the code inside the while loop is run over this list.

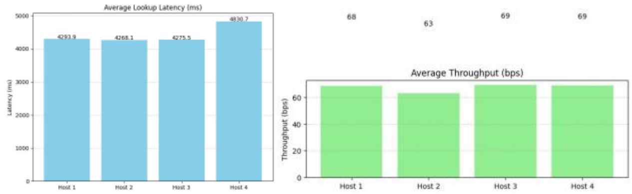
Finally, after the final IP address is obtained, elapsed_time is calculated and the data is returned.

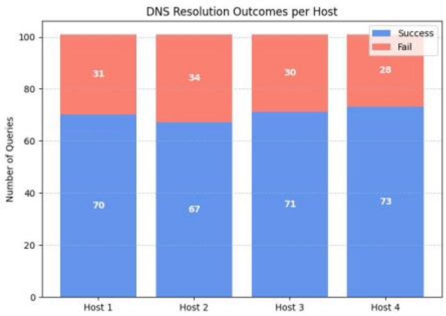
Observations and Analysis

I. Plotting the graphs from Part B.

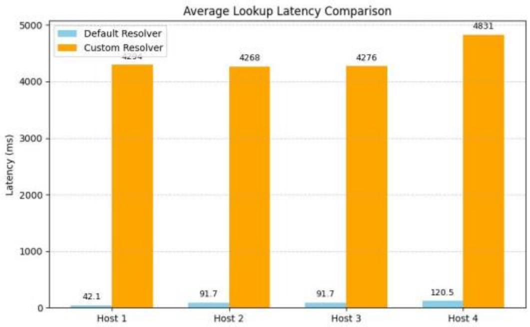


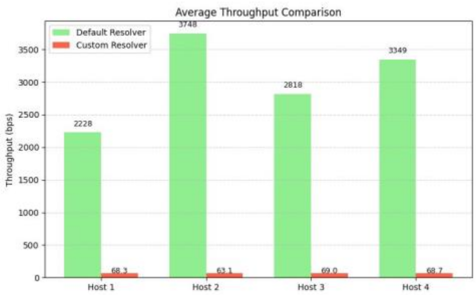
II. Part D (Along with multiserver and cache)

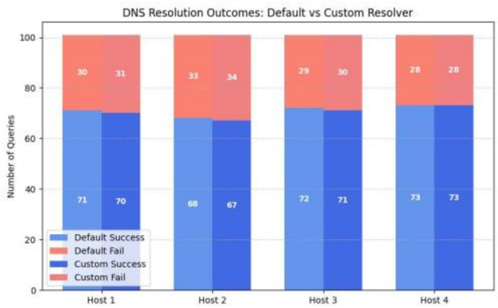




III. Comparisons between Part B and Part D







During the comparison it is clearly visible that the Default host resolver has much **lower latency**. This could be because that our function in default resolver is a true iterative solver, while in first case it could be because systemd resolver might directly pinging the recursive resolver. Also in our case we have to ping various IPs which might be heavy because of congestion in the network.
It was also visible during the run time, which in the case of default resolver was in seconds for the entire PCAP file, while in case of custom resolver it took more than 10 minutes. We can also see that, the resolutions had slighly decreased in case of Custom resolvers.

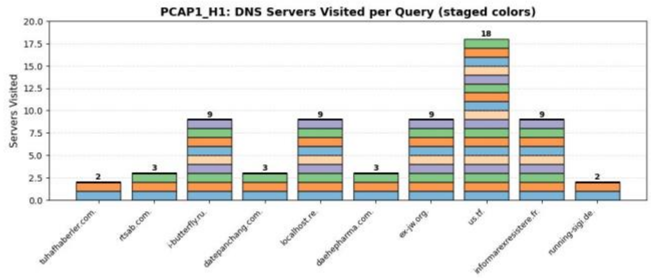
Throughput was **much less** in custom resolver, as it is inversely related to total resolution time. In some cases this relation has failed while running because failure of query can deviate, as in the calculation only included success for count.

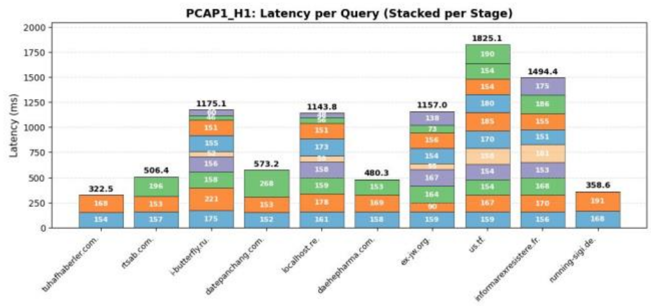
Success achieved was slightly low in case where cache and multi-server querying was not there as in this case failure even at single stage could cause to mark the domain as failure.

After the implementation of cache, there was slight improvement in success while increasing the latency due to reduction in timeout. After the Multi-server (iterating over the server list), there was improvement in success almost nearing the results of the custom resolver.

Differences can be seen in the tables attached.

IV. Analysis for the first 10 queries of PCAP_1 file





First graphs analyze the number of servers visited for the final resolution. It can be seen in the fluctuations, that in some cases because of early failure in the code, the servers visited are either too less, and in some case because of late resolution after iterating through the entire list, an answer or timeout occurred.

Second graph tells the query latency for each of the request made. It can be seen that it also fluctuates along with servers.

Note: JSON files are attached in the github containing the data from a – i in Question D.

Also, for queries, JSON file has logged multiple times. This happens because of the internal timeout in case the responses in not received soon by the host. The custom_resolver was queried multiple times again for the same domain_name.