Computer Networks Assignment 2

Akash Gupta 23110020 | Kaushal Bule 23110160

GitHub - https://github.com/Kaushal845/CN-Assn2

- ## Setup

For this assignment, we used **Oracle VirtualBox**, which was already available on the computers in the lab. We began by **installing Mininet on the system** using Git, allowing us to create and test network setups in a virtual environment.

```
student@student-VirtualBox:~/Downloads/CN-A2$ cd /tmp && git clone https://github.com/mininet/mininet.git && cd mininet && sudo ./util/install.sh -a
```

- ## Part A

To create and test the required network topology, we used the **custom_topo.py** script. When this script is executed, the network is **initialized**, all **components are configured**, **interfaces are assigned**, and the **controller is** automatically activated.

```
student@student-VirtualBox:~/Downloads/CN-A2$ sudo python3 custom_topo.py
```

```
student@student-VirtualBox:~/Downloads/CN-A2$ sudo python3 custom_topo.py
*** Adding controller
*** Adding switches
*** Adding hosts
```

The script then **creates the desired topology**. After that, it **tests network connectivity** using the **pingall** command to ensure all hosts can communicate. Additionally, we perform a **pairwise ping test** between **host h1** and all other hosts to verify individual connections in detail.

```
*** Creating links (bw=100Mbps, specified delays)
(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 5ms delay) (100.00Mbit 5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 8ms delay
) (100.00Mbit 10ms delay) (100.00Mbit 10ms delay) (100.00Mbit 1ms delay) (100.00Mbit 1ms 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 5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 1ms
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 5ms delay) (100.00Mbit 8ms delay) (100.00Mbit 1ms delay) (10
0.00Mbit 2ms delay) (100.00Mbit 8ms delay) (100.00Mbit 10ms delay) (100.00Mbit 2ms delay) (100.00Mbit 10ms delay)
```

```
*** Running tests
*** Ping all hosts
*** 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)
```

The output confirms that **all nodes are fully reachable**, indicating that the **switches are correctly interconnected**, **IP addresses are properly assigned**, and there are **no configuration errors** in the network setup.

```
*** Ping specific pairs (h1->h2 ,h1->h3, h1->h4 and h1->dns)
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=23.7 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=23.2 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=22.9 ms

--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 22.876/23.246/23.659/0.321 ms
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=42.6 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=52.3 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=54.0 ms

--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2008ms
rtt min/avg/max/mdev = 42.553/49.628/54.010/5.050 ms
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=59.7 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=64.6 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=69.9 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 59.725/64.725/69.884/4.148 ms
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=19.5 ms
64 bytes from 10.0.0.5: icmp_seq=2 ttl=64 time=21.9 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=53.6 ms

--- 10.0.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2009ms
rtt min/avg/max/mdev = 19.531/31.688/53.600/15.524 ms
```

Next, we **measured the TCP throughput** between **host h1** and **host h4** using the **iperf** tool. This helped us evaluate the **data transfer performance** and verify the efficiency of the established network connection.

```
*** Run iperf between h1 and h4 (TCP)
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.1 port 51558 connected with 10.0.0.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 5.3 sec  39.8 MBytes  63.2 Mbits/sec
```

The measured throughput of **63.2 Mbps** indicates a **stable TCP connection** over a **network path with a delay**. However, it is **lower than the theoretical 100 Mbps limit** due to several factors, including **TCP handshake and acknowledgment delays**, **queuing delays**, and **Mininet's internal processing overhead**. These factors collectively contribute to the reduction in achievable throughput compared to the ideal scenario.

- **Part B**

We began by **downloading all the PCAP files** from the provided link. These files were then processed using **tshark** to **extract and filter DNS queries**, which were saved in a file named **Hi_domains_cleaned.txt**. This was done using the following command:

```
tshark -r PCAP_1_H1.pcap -Y 'dns.qry.name matches ".*\..*" and !(dns.qry.name contains "wpad") and !(dns.qry.name contains "isatap") and !(dns.qry.name contains "isilon") and !(dns.qry.name contains "localhost")' -T fields -e dns.qry.name > H1_domains_cleaned.txt
```

Next, we executed the **dns_topo.py** script, which **instantiates the network topology** with predefined **bandwidth** and **delay parameters** using the topology class.

To enable DNS queries to reach beyond the simulated environment, **Network Address Translation (NAT)** was configured using the **addNATandInternet()** function. The NAT node was **connected to Switch**

S2, providing all hosts with **outbound Internet access**. If the **local DNS resolver (10.0.0.5)** failed to respond, the hosts automatically attempted resolution using **Google's public DNS (8.8.8.8)** through the NAT interface. This **dual-resolution setup** effectively simulates **redundant DNS resolution**, improving **network resilience** and reducing **latency during failures**.

We then used **Linux's nslookup utility** to perform DNS resolutions. During these operations, we recorded several key performance metrics, including **average latency**, **throughput**, **success rate**, and **failure rate**.

Finally, a separate **result text file** was generated for each host. The **last line** of each file contained an **aggregate summary** of the recorded values—namely, the **total queries**, **successful and failed resolutions**, **average latency**, and **throughput**.

The results are as follows:

Host	Total Queries	Success	Fail	Avg Latency (ms)	Throughput (q/s)
H1	105	75	30	536	2.28
H2	106	72	34	547	2.30
H3	106	72	34	542	2.30
H4	106	77	29	614	2.00

- **Part C**

In this section, we first **created the specified network topology** and introduced a **custom DNS server host** with the IP address **10.0.0.5**. All other hosts were then **configured to use this server as their primary**

nameserver, ensuring that all **DNS queries** were directed to the **local resolver** instead of external DNS servers such as **Google's 8.8.8.8**.

Each host was configured to use the custom resolver by executing the command in the script:

```
host.cmd('echo "nameserver 10.0.0.5" | tee /etc/resolv.conf')
```

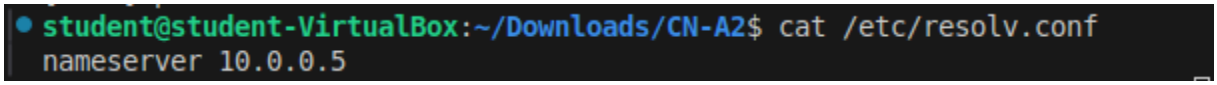
Initially, we attempted to achieve this using the command:

```
host.cmd('echo "nameserver 10.0.0.5" > /etc/resolv.conf')
```

However, this approach **failed silently** because the file **/etc/resolv.conf** is **write-protected**, preventing direct modification. The use of `tee` successfully bypassed this issue, allowing the DNS configuration to be applied correctly across all hosts.

The commands were run in the t32.py script.

This can be verified using the following command



This confirms the successful DNS reassignment to 10.0.0.5, as shown above.

We tested reachability using the pingAll command. We also do pairwise ping tests between h1 and h4, DNS. We also run iperf between h1 and h4.

```
*** 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)
*** Ping specific pairs (h1->h4 and h1->dns)
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=59.2 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=62.8 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=73.0 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 59.178/64.982/72.950/5.826 ms
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=28.5 ms
64 bytes from 10.0.0.5: icmp_seq=3 ttl=64 time=21.5 ms

--- 10.0.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 19.628/23.188/28.484/3.817 ms
*** Run iperf between h1 and h4 (TCP)
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.1 port 52554 connected with 10.0.0.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 5.2 sec  43.1 MBytes  69.5 Mbits/sec
```

- **Part D**

In this part, we first downloaded Mininet VM and VirtualBox. To simplify interaction with the Mininet environment, I used SSH access from my Windows operating system.

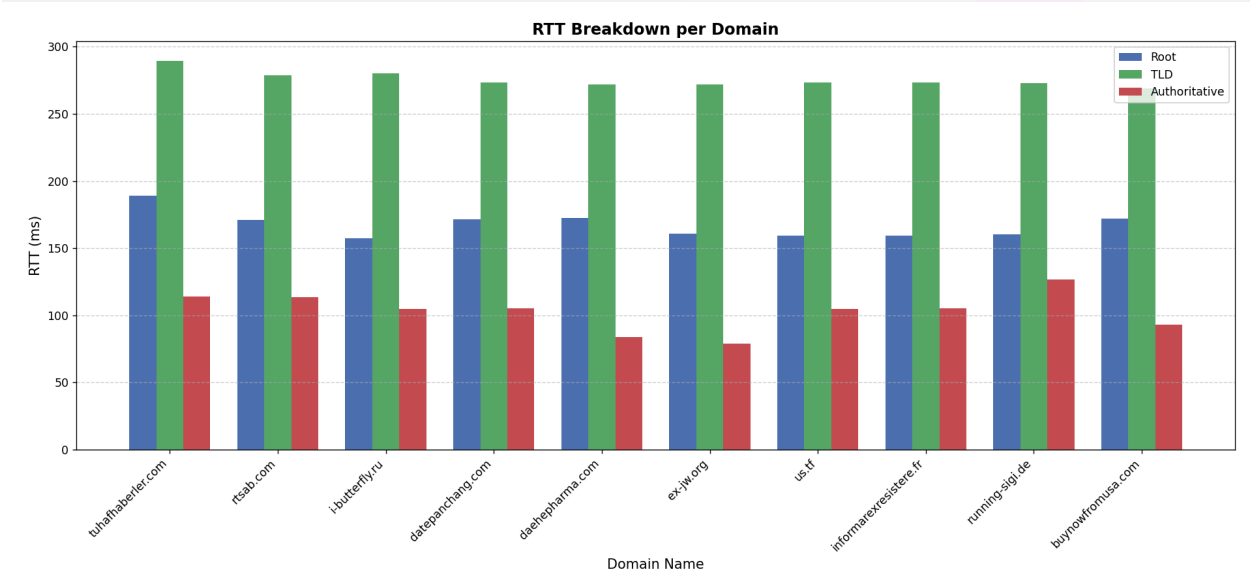
In this part, we first create the topology, add a custom DNS resolver (10.0.0.5), and NAT for internet access using dns_topo.py.

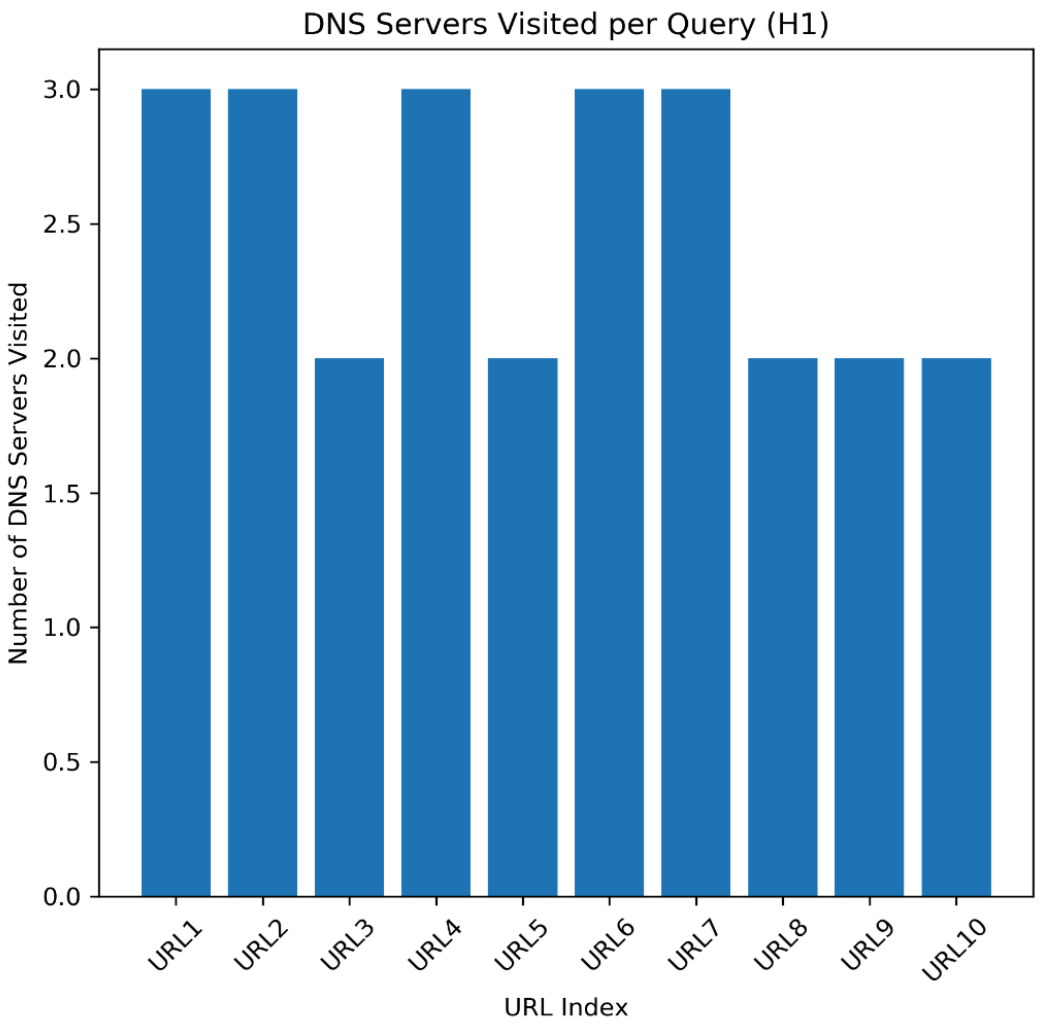
Network Address Translation (NAT) is a networking technique used to modify the **source or destination IP addresses** of packets as they pass through a router or gateway. It allows multiple devices within a private network to **share a single public IP address** for accessing external networks such as the Internet.

NAT improves **security** by hiding internal IP addresses from external entities and helps **conserve IPv4 addresses** by enabling reuse of private IP ranges. In addition, NAT can be configured to support **port forwarding**, allowing specific inbound traffic to reach designated internal hosts.

In network simulations or setups (like Mininet), NAT is often used to **provide Internet access** to virtual hosts while keeping their internal addresses private.

Here is a plot representing the Latency/RTT of the first 10 domains in each of the steps -






```
[106/106] florenseusa.com -> 34.95.203.179 (187.2 ms)

=== Summary ===
Total queries: 106
Successful: 73
Failed: 33
Average latency: 294.60 ms

Results saved to /home/mininet/ass_2/h44_results.csv
```

Comparison and Analysis

The **custom DNS resolver (Part D)** demonstrates mixed performance compared to the **default system resolver (Part B)**.

1. Success and Failure - Both resolvers showed similar reliability, with about 69–73% successful lookups. In Part D, there were 73 successes out of 106 queries (69%), close to Part B's 70% average. The few extra failures in the custom resolver were due to its strict 2-second timeout and single query path without retries or caching, while the system resolver performed slightly better thanks to retries and cached responses.
2. Latency - The average latency decreased notably from 540–610 ms in Part B to 295 ms in Part D, indicating faster overall query resolution. This improvement is mainly due to fewer network hops and lower logging overhead in the iterative resolver. However, since Part D performs full stepwise resolution (Root, then TLD, and then Authoritative), some domains may still experience higher latency.
3. Throughput - Throughput, being inversely related to latency, was slightly higher in Part D (3.4 q/s) compared to Part B (2.0–2.3 q/s). This shows more efficient query handling without cache dependency, though overall performance still depends on network stability and host load.