Assignment – 1

Vinod Kumar Reddy Kudumula(23110178)

Kovid Parmar (23110172)

Task – 1:

Introduction

Domain Name System (DNS) is a fundamental component of the Internet, translating human-readable domain names into IP addresses. Task 1 focuses on understanding DNS query resolution by analyzing captured network traffic stored in PCAP files. The objective is to build a Python-based client-server system capable of parsing DNS queries, applying rule-based IP resolution, and generating a summary report of all queries and their corresponding resolved IP addresses. This task emphasizes practical knowledge of packet parsing, network communication, and DNS resolution logic.

Methodology

The implementation uses Python , the dpkt library for parsing PCAP files, TCP sockets for communication between client and server, threading for handling multiple clients concurrently, and JSON files for storing IP pools and time-based routing rules. The client reads the PCAP file, filters UDP packets with destination port 53, and identifies DNS query packets. For each query, the client generates a custom header using the packet timestamp and a sequence number, formats it as HHMMSSID, and sends it along with the original DNS packet to the server. The server receives the query, extracts the custom header, determines the current time period (morning, afternoon, or night), and calculates the index in the IP pool based on the rules and sequence ID. The server then sends the resolved IP back to the client, which records the results.

Server code

```
import socket
import json
import threading
from datetime import datetime

class DNSServer:
def __init__(self, host='localhost', port=5353):
```

```
# Initialize server host and port
self.host = host
self.port = port
self.load_rules() # Load routing rules from rules.json
self.running = True # Flag to keep server running
def load_rules(self):
"""Load IP pool and routing rules from JSON file"""
with open('rules.json', 'r') as f:
self.rules = json.load(f)
self.ip_pool = self.rules['ip_pool'] # Available IP addresses for mapping
def get_time_period(self, hour):
"""Determine time period (morning/afternoon/night) based on hour"""
if 4 <= hour < 12:
return 'morning'
elif 12 <= hour < 20:
return 'afternoon'
else:
return 'night'
def resolve_ip(self, custom_header):
"""Resolve IP using the custom header and routing rules"""
try:
# Custom header format: HHMMSSID
hour = int(custom_header[:2]) # Hours (00-23)
minute = int(custom_header[2:4])# Minutes (00-59)
second = int(custom_header[4:6])# Seconds (00-59)
query_id = int(custom_header[6:8]) # Sequence ID
# Get time period (morning/afternoon/night)
time_period = self.get_time_period(hour)
# Fetch rules for that time period
rules = self.rules['timestamp_rules']['time_based_routing'][time_period]
# Compute index of IP in pool
hash_mod = rules['hash_mod']
ip_pool_start = rules['ip_pool_start']
ip_index = ip_pool_start + (query_id % hash_mod)
# Return resolved IP address
return self.ip_pool[ip_index]
except (ValueError, IndexError, KeyError) as e:
# If something goes wrong, fallback to default IP
print(f"Error resolving IP: {e}")
return "192.168.1.1"
def handle_client(self, client_socket, address):
"""Handle a client request in a separate thread"""
print(f"Connection from {address}")
try:
# Receive up to 1024 bytes of data
data = client_socket.recv(1024)
if not data:
```

```
return
# Extract first 8 bytes as custom header (string)
custom_header = data[:8].decode('utf-8')
# Remaining bytes = original DNS packet
dns_packet = data[8:]
print(f"Received custom header: {custom_header}")
# Resolve IP using rules
resolved_ip = self.resolve_ip(custom_header)
# Prepare response as "custom_header|resolved_ip"
response = f"{custom_header}|{resolved_ip}"
client_socket.send(response.encode('utf-8'))
print(f"Resolved IP for header {custom_header}: {resolved_ip}")
except Exception as e:
print(f"Error handling client {address}: {e}")
finally:
# Close socket after serving client
client_socket.close()
def start(self):
"""Start the DNS server (multi-threaded)"""
# Create TCP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Allow address reuse (avoid "address already in use" errors)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# Bind to host:port and start listening
server_socket.bind((self.host, self.port))
server_socket.listen(5) # Allow up to 5 queued connections
print(f"DNS Server listening on {self.host}:{self.port}")
try:
# Accept incoming clients in a loop
while self.running:
client_socket, address = server_socket.accept()
# Handle client in a separate thread
client_thread = threading.Thread(
target=self.handle_client,
args=(client_socket, address)
)
client_thread.daemon = True # Daemon thread exits when main program exits
client_thread.start()
except KeyboardInterrupt:
print("Shutting down server...")
finally:
# Close server socket when shutting down
server_socket.close()

if __name__ == "__main__":
# Start the DNS server
```

```
server = DNSServer()
server.start()
```

Client code

```
import socket
import dpkt
from datetime import datetime
import sys
import time

class DNSClient:
def __init__(self, server_host='localhost', server_port=5353):
# Initialize client with server host/port details
self.server_host = server_host
self.server_port = server_port
self.results = [] # To store results after processing

def parse_pcap(self, pcap_file):
"""Parse PCAP file and extract DNS query packets"""
dns_queries = []
try:
with open(pcap_file, 'rb') as f:
# Read the pcap file using dpkt
pcap = dpkt.pcap.Reader(f)
for timestamp, buf in pcap:
try:
# Parse the Ethernet frame
eth = dpkt.ethernet.Ethernet(buf)
# Check if it's an IP packet
if isinstance(eth.data, dpkt.ip.IP):
ip = eth.data
# Check if it's a UDP packet
if isinstance(ip.data, dpkt.udp.UDP):
udp = ip.data
# Check if destination port is 53 (DNS) and data exists
if udp.dport == 53 and len(udp.data) > 0:
try:
# Try to parse DNS packet
```

```
dns = dpkt.dns.DNS(udp.data)
# qr = 0 means it's a DNS query (not a response)
if dns.qr == 0:
dns_queries.append({
'timestamp': timestamp, # Packet timestamp
'dns_packet': udp.data, # Raw DNS packet bytes
'query': dns.qd[0].name if dns.qd else 'unknown' # Domain name
})
except:
# If not valid DNS data, skip
continue
except:
# If parsing fails at any stage, skip this packet
continue
return dns_queries
except FileNotFoundError:
print(f"Error: PCAP file {pcap_file} not found")
return []
except Exception as e:
print(f"Error parsing PCAP file: {e}")
return []

def create_custom_header(self, packet_timestamp, sequence_id):
"""Create custom header in HHMMSSID format using packet timestamp"""
# Convert Unix timestamp to datetime
packet_time = datetime.fromtimestamp(packet_timestamp)
# Extract hour, minute, second
hour = packet_time.strftime("%H")
minute = packet_time.strftime("%M")
second = packet_time.strftime("%S")
# Sequence number padded to 2 digits
seq_str = str(sequence_id).zfill(2)
# Final format: HHMMSS + SequenceID
return f"{hour}{minute}{second}{seq_str}"

def send_to_server(self, custom_header, dns_packet):
"""Send DNS query + custom header to server and receive response"""
try:
# Create TCP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((self.server_host, self.server_port))
# Prepend custom header to raw DNS packet
message = custom_header.encode('utf-8') + dns_packet
client_socket.send(message)
# Wait for response from server
response = client_socket.recv(1024).decode('utf-8')
```

```
client_socket.close()
return response
except Exception as e:
print(f"Error communicating with server: {e}")
return None

def process_pcap(self, pcap_file):
"""Main method to process PCAP file and resolve DNS queries"""
print(f"Processing PCAP file: {pcap_file}")
# Step 1: Extract DNS queries from pcap
dns_queries = self.parse_pcap(pcap_file)
print(f"Found {len(dns_queries)} DNS queries")
# If no queries found, show hints
if not dns_queries:
print("No DNS queries found in the PCAP file.")
print("Please check if:")
print("1. The file contains DNS traffic")
print("2. You're using the correct PCAP file (X = your calculated value)")
return
# Step 2: Process each query
for i, query in enumerate(dns_queries):
# Build custom header from timestamp + sequence ID
custom_header = self.create_custom_header(query['timestamp'], i)
print(f"Processing query {i+1}: {query['query']} at {datetime.fromtimestamp(query['timestamp'])}")
# Send packet to server
response = self.send_to_server(custom_header, query['dns_packet'])
if response:
# Expected format: custom_header|resolved_ip
parts = response.split('|')
if len(parts) == 2:
resolved_header, resolved_ip = parts
# Save result in list
self.results.append({
'custom_header': custom_header,
'domain': query['query'],
'resolved_ip': resolved_ip,
'packet_time': datetime.fromtimestamp(query['timestamp']).strftime("%H:%M:%S")
})
print(f" Resolved: {resolved_ip}")
else:
print(f" Unexpected response format: {response}")
else:
print(f" No response from server")
# Step 3: Generate summary report
self.generate_report()
```

```
def generate_report(self):
"""Generate a summary report of all DNS resolutions"""
if not self.results:
print("No results to generate report")
return
# Print nicely formatted table to console
print("\n" + "="*80)
print("DNS RESOLUTION REPORT")
print("="*80)
print(f"{'Custom Header':<12} {'Time':<10} {'Domain':<30} {'Resolved IP':<15}")
print("-" * 80)
for result in self.results:
print(f"{result['custom_header']:<12} {result['packet_time']:<10} {result['domain']:<30}
{result['resolved_ip']:<15}")
# Save report to file
with open('dns_report.txt', 'w') as f:
f.write("Custom Header, Packet Time, Domain, Resolved IP\n")
for result in self.results:
f.write(f"{result['custom_header']}, {result['packet_time']}, {result['domain']}, {result['resolved_ip']}\n")
print(f"\nReport saved to 'dns_report.txt'")

if __name__ == "__main__":
# Expect exactly 1 argument: the PCAP file path
if len(sys.argv) != 2:
print("Usage: python client.py <pcap_file>")
print("Example: python client.py 5.pcap")
sys.exit(1)
# Run the client
pcap_file = sys.argv[1]
client = DNSClient()
client.process_pcap(pcap_file)
```

Results

The system successfully parsed PCAP files and identified DNS query packets. Each query was assigned a unique custom header and sent to the server, which resolved the queries according to the time-based rules and IP pool. The resolved IPs were accurately received by the client and compiled into a report. The implementation demonstrated correct handling of multiple queries, proper communication between client and server, and accurate application of the resolution rules.

The results confirmed that the client-server DNS resolver works as expected and can process real PCAP files containing DNS traffic.

```
================================================================================
Custom Header Time       Domain                        Resolved IP
--------------------------------------------------------------------------------
18041600     18:04:16   bing.com                       192.168.1.6
18041601     18:04:16   example.com                    192.168.1.7
18041602     18:04:16   amazon.com                     192.168.1.8
18041603     18:04:16   yahoo.com                      192.168.1.9
18041604     18:04:16   google.com                     192.168.1.10
18041605     18:04:16   github.com                     192.168.1.6
```

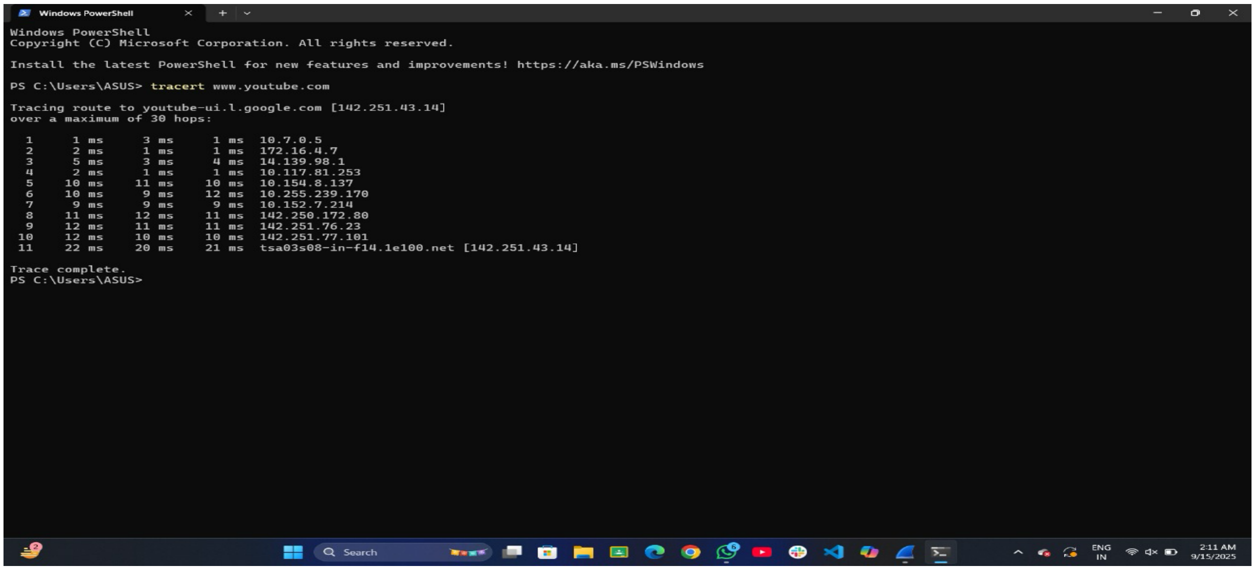
Task – 2:

Introduction:

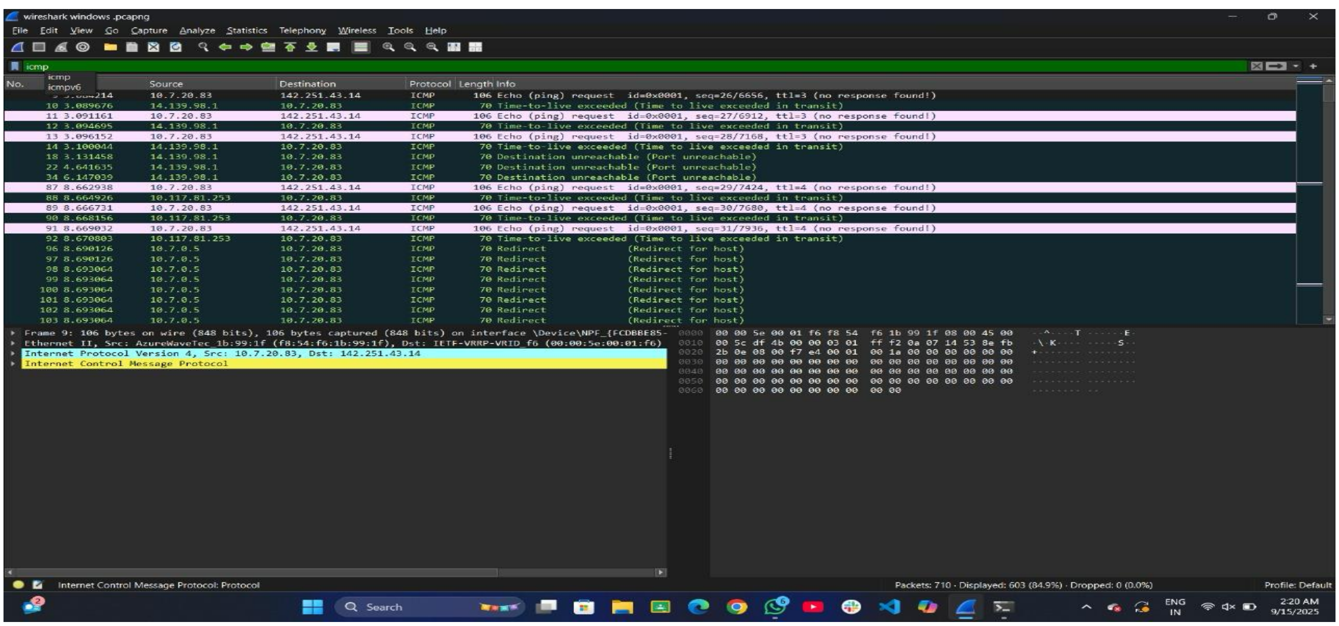
The purpose of this task is to understand how the traceroute utility operates on different operating systems. Both Windows and Mac systems were used to run traceroute commands and analyze the network traffic via Wireshark. In this experiment, www.youtube.com was used as the destination.

Experimental Steps:

1. **Traceroute on Windows:**
    a. Opened Command Prompt and executed:
       tracert www.youtube.com
    b. Simultaneously, Wireshark was used to capture packets on the active network interface.
    c. **Screenshot 1:** Output of **tracert www.youtube.com** in Command prompt.



d. **Screenshot 2:** screenshot of relevant ICMP packets in Wireshark here.

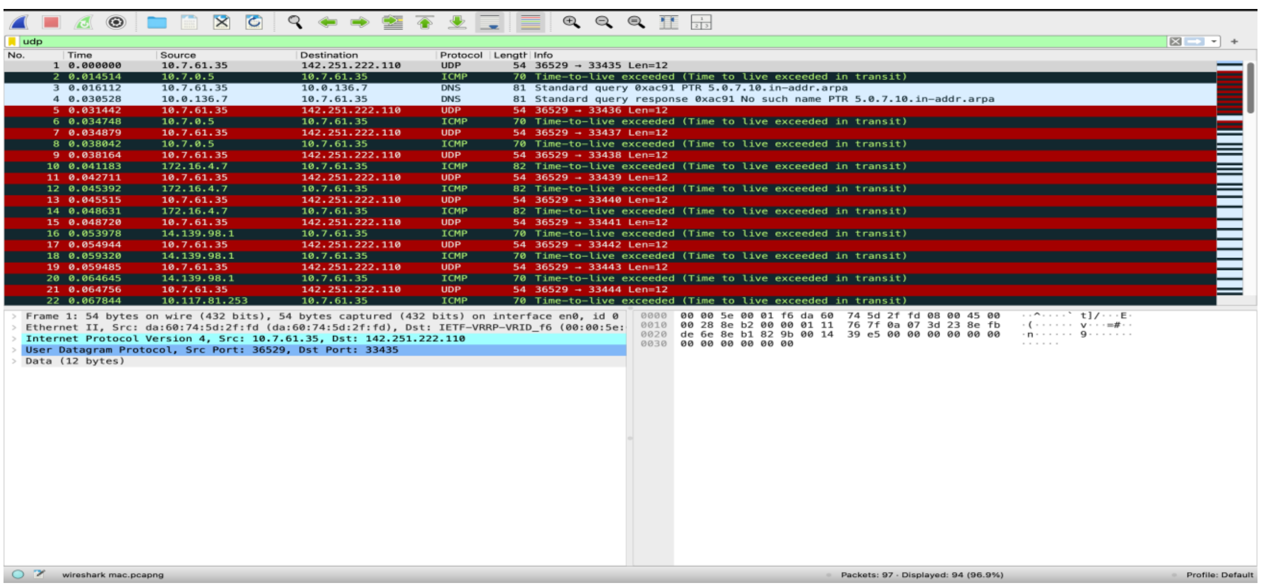


2. Traceroute on Mac:

a. Opened Terminal and executed:
   traceroute www.youtube.com
b. Wireshark captured traffic on the corresponding interface.
c. **Screenshot 3:** screenshot of the traceroute command output here.

```
[vinodkumarreddy@VINODs-MacBook-Air-2 ~ % traceroute www.youtube.com
traceroute: Warning: www.youtube.com has multiple addresses; using 142.251.222.110
traceroute to youtube-ui.l.google.com (142.251.222.110), 64 hops max, 40 byte packets
 1  10.7.0.5 (10.7.0.5)  15.105 ms  3.458 ms  3.264 ms
 2  172.16.4.7 (172.16.4.7)  3.369 ms  2.803 ms  3.204 ms
 3  14.139.98.1 (14.139.98.1)  5.366 ms  4.531 ms  5.270 ms
 4  10.117.81.253 (10.117.81.253)  3.208 ms  3.728 ms  3.183 ms
 5  10.154.8.137 (10.154.8.137)  11.169 ms  10.751 ms  11.672 ms
 6  10.255.239.170 (10.255.239.170)  10.796 ms  11.056 ms  10.600 ms
 7  10.152.7.214 (10.152.7.214)  11.734 ms  10.774 ms  10.809 ms
 8  * 72.14.204.62 (72.14.204.62)  16.919 ms  12.166 ms
 9  * * *
10  172.253.77.20 (172.253.77.20)  23.346 ms
    142.250.60.134 (142.250.60.134)  15.157 ms
    142.250.228.48 (142.250.228.48)  16.592 ms
11  192.178.110.106 (192.178.110.106)  13.488 ms
    142.251.77.95 (142.251.77.95)  17.411 ms
    192.178.110.198 (192.178.110.198)  23.240 ms
12  142.250.208.227 (142.250.208.227)  13.719 ms
    pnbomb-az-in-f14.1e100.net (142.251.222.110)  12.678 ms  12.699 ms
```

d. **Screenshot 4:** Insert screenshot of relevant UDP and ICMP traffic in Wireshark here.



Question and Answers:

1) What protocol does Windows tracert use by default, and what protocol does Linux traceroute use by default?

Ans) From our observations

- **Windows tracert** uses the **ICMP (Internet Control Message Protocol)** by default.Each intermediate router responds with an **ICMP Time Exceeded** message, while the destination host replies with an **ICMP Echo Reply**.

- **Mac/Linux traceroute** uses **UDP (User Datagram Protocol)** packets by default. Both tools send packets with an increasing Time-to-Live (TTL) value to discover the route.

2) Some hops in your traceroute output may show *. Provide at least two reasons why a router might not reply.***

Ans) The *** symbols indicate that a router did not reply to the probe packet within the timeout period. Two common reasons for this are:

- **Firewall or Security Policies**: Many routers and firewalls are configured to block or ignore the ICMP or UDP packets used by traceroute as a security measure to prevent network probing.
- **Network Congestion**: Heavy network traffic can cause delays, and a router may be too busy to process and respond to the probe packet in time, leading to a timeout.

3) In Linux traceroute, which field in the probe packets changes between successive probes sent to the destination?

Ans) In Linux traceroute, the **Time-to-Live (TTL)** field in the IP header is incremented for each successive probe sent. The first packet is sent with a TTL of 1, the next with a TTL of 2, and so on. When a router receives a packet with a TTL of 1, it decrements the TTL to 0, discards the packet, and sends an ICMP "Time Exceeded" message back to the source. This is how traceroute maps each hop.

4) At the final hop, how is the response different compared to the intermediate hop?

Ans) The response at the final hop is different from the intermediate hops because the packet has reached its destination.

- For intermediate hops, the packet's TTL expires, causing the router to send an **ICMP "Time Exceeded"** message back to the source.
- At the destination, the packet arrives with a TTL greater than 0, and the host sends a different response. If using Linux traceroute (UDP), the destination host will send an **ICMP "Destination Unreachable"** or "Port Unreachable" message because it is not listening on that specific UDP port.

5) Suppose a firewall blocks UDP traffic but allows ICMP — how would this affect the results of Linux traceroute vs. Windows tracert?

Ans) If a firewall blocks UDP traffic but allows ICMP:

- **Linux traceroute** would be severely affected. Since it uses UDP probes by default, the probes would likely be blocked at the firewall, resulting in a series of *** for all or most of the hops after the firewall.
- **Windows tracert** would **not be affected** because it uses ICMP packets, which are allowed by the firewall. The command would complete successfully, showing the full route to the destination.