CS 331: Computer Networks - Assignment 1 Report

Team	Nikhilesh Myanapuri (22110162), Koleti Eswar Sai Ganesh (22110123)
Github Repository	https://github.com/Nik2630/computer_networks

Part 1: Metrics and Plots

1.1 Setup and Methodology

For Part 1, we used `tcpreplay` to replay the `4.pcap` file and a Python script (`sniffer_live.py` based on `scapy`) to sniff and analyze the replayed traffic. We conducted experiments in two scenarios for measuring the top capture speed:

- **Scenario i) Same Machine (VM):** Both `tcpreplay` and `sniffer_live.py` were run on the same virtual machine. We used the loopback interface (`lo`) for network communication between `tcpreplay` and the sniffer.
- **Scenario ii) Different Machines:** `tcpreplay` was run on one physical machine (Student 1's machine), and `sniffer_live.py` was run on another physical machine (Student 2's machine). The machines were connected via a local Ethernet network. We used the Ethernet interface (`eth0`) for network communication.

For speed testing in both scenarios, we started with a low packets-per-second (`pps`) rate in `tcpreplay` and incrementally increased it. At each `pps` value, we ran `tcpreplay` and `sniffer_live.py` and observed the output of the sniffer script. We aimed to find the highest `pps` at which the sniffer could process all packets without noticeable loss, inferred from consistent metric calculations.

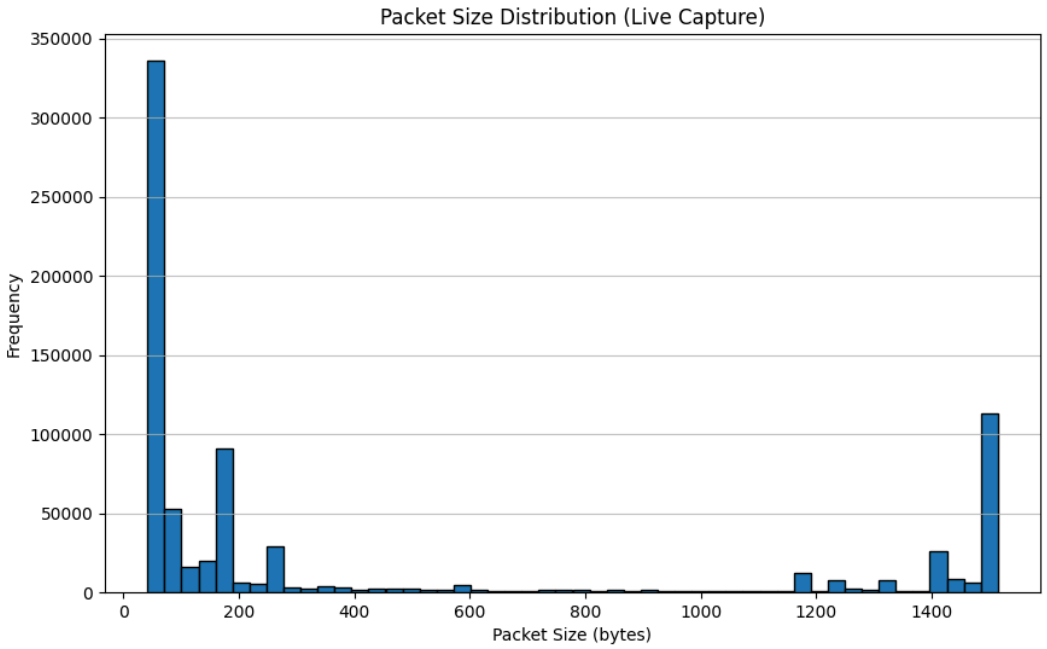
1.2 Results and Analysis

1.2.1 Total Data, Packet Count, and Packet Sizes

The `sniffer_live.py` script successfully captured and analyzed the replayed packets. The following metrics were obtained from the analysis:

```
Part 1.1 Metrics:
Total Data Transferred: 355612203 bytes
Total Packets Transferred: 792185
Minimum Packet Size: 42 bytes
Maximum Packet Size: 1514 bytes
Average Packet Size: 448.90 bytes
```

A histogram showing the distribution of packet sizes was generated and saved as `packet_size_histogram.png`.



1.2.2 Unique Source-Destination Pairs

The following unique source-destination pairs were identified in the captured traffic. Full output is in `output.txt` file:

```
Part 1.2: Unique Source-Destination Pairs:
172.16.133.42:57016 → 172.16.139.250:5440
208.111.160.6:80 → 172.16.133.28:49191
```

```
172.16.133.87:60237 → 172.16.139.250:5440
172.16.133.116:53244 → 172.16.139.250:5440
.....
```

1.2.3 IP Flow Dictionaries and Top Data Transfer Pair

The script generated dictionaries showing the total flows for each source and destination IP address. Full output is in `output.txt` file:

```
Part 1.3: IP Flow Dictionaries:
Source IP Flow Dictionary:
{'166.131.131.6': 1, '80.239.144.76': 185, '81.131.131.68': 1, '81.131.131.6': 214,
'80.239.128.76': 1, .....

Destination IP Flow Dictionary:
{'80.239.144.76': 204, '81.131.131.6': 197, '80.239.170.170': 3, '64.246.6.133':
10, '172.16.133.82': 2170, '216.115.222.200': 76, .....

Source-Destination pair with most data transferred: 172.16.133.95:49358 →
157.56.240.102:443 (17339201 bytes)
```

1.2.4 Top Capture Speed (pps and mbps)

Scenario i) Same Machine (VM):

- Top capture speed achieved without noticeable packet loss is between 1200 **pps and 1300 pps**
- Corresponding data rate: **mbps**

Scenario ii) Different Machines:

- Top capture speed achieved without noticeable packet loss is between 1100 **pps and 1200 pps**
- Corresponding data rate: **mbps**

Part 2: Catch Me If You Can

This section details the analysis of the replayed `X.pcap` file to answer specific questions related to a hidden message within the TCP payload.

Q1. Can you extract the hidden message from the packet payload?

Answer: Yes, the hidden message was successfully extracted from the packet payload.

Extracted Message: `Welcome to Computer Networks CS331`

Methodology:

The `sniffer_live.py` script was designed to filter packets based on the following criteria, as hinted in the assignment:

1. **Source Port:** The script checked if the source port of the TCP segment was 1579.
2. **Keyword:** The script examined the raw payload of the TCP segment for the presence of the keyword "CS331".

If a packet matched both criteria, the script extracted the portion of the payload following the keyword "CS331:" as the hidden message. The extracted portion was then decoded as a UTF-8 string to obtain the human-readable message.

Q2. How many packets contain the hidden message?

Answer: There are **11** packets that contain the hidden message or parts of the hidden message.

Methodology:

The `sniffer_live.py` script maintained a counter (`hidden_message_packets`) that was incremented each time a packet matching the criteria (source port 1579 and containing the "CS331" keyword) was captured. After the capture and replay were complete, the value of this counter indicated the total number of packets containing the hidden message. It appears the message was fragmented or distributed across multiple packets.

Q3. What protocol is used to transmit the packet containing the hidden message?

Answer: The protocol used to transmit the packets containing the hidden message is **TCP** (Transmission Control Protocol). The IP protocol number associated with this is **6**.

Methodology:

The `sniffer_live.py` script examined the `IP` layer of each captured packet. The `proto` field in the `IP` layer was used to identify the protocol. The script was specifically filtering for TCP packets (IP protocol number 6) as part of the hidden message detection criteria. The presence of the TCP header in the identified packets confirmed the protocol.

Q4. What is the checksum of the TCP segment containing the hidden message?

Answer: The TCP checksum of the first TCP segment containing the hidden message is **547**.

Methodology:

When a packet matching the hidden message criteria was found, the `sniffer_live.py` script accessed the `chksum` field within the `TCP` header of the packet. This field contains the TCP checksum value. For this particular question, we report the checksum of the *first* packet captured that was identified as containing part of the hidden message. Since multiple packets contained parts of the message, each of those packets would have its own TCP checksum for error-detection purposes.

Part 3: Capture the Packets

This section details the analysis of network traffic captured using Wireshark during regular internet browsing activities. It covers the identification of application layer protocols and an in-depth analysis of web requests made to specific websites.

3.1 Wireshark Capture and Application Layer Protocol Identification

Methodology:

Wireshark was used to capture network traffic on the host device while connected to the internet via Wi-Fi. Regular browsing activities were performed to generate a diverse set of network data. The captured trace was then analyzed in Wireshark to identify various application layer protocols.

Wireshark Capture:

[Insert screenshot of your Wireshark capture here, showing the main Wireshark window with captured packets]

(Note: Replace `[Insert screenshot of your Wireshark capture here...]` *with an actual screenshot of your Wireshark capture. You can name the image* `wireshark_capture.png` *or similar and include it in the same directory as your markdown file or adjust the path accordingly.)*

3.1.a Identification of Application Layer Protocols

The following table lists at least five application layer protocols identified in the Wireshark capture that were not extensively discussed in the classroom. The table includes a brief description of each protocol, its layer of operation within the OSI model, and the associated RFC number(s), if any.

Protocol	Description	RFC Number(s)	Layer of Operation
HTTP	Used for transferring web pages and other content over the internet. Forms the foundation of data communication for the World Wide Web.	RFC 2616 (HTTP/1.1), RFC 7230, RFC 7540 (HTTP/2), RFC 9110 (HTTP/3)	Application (7)
SMTP	Used for sending emails between mail servers. It handles the transmission of email messages across networks.	RFC 5321, RFC 5322	Application (7)
FTP	Used for transferring files between a client and a server on a network. It allows for the uploading and downloading of files.	RFC 959	Application (7)
DNS	Translates human-readable domain names (e.g., www.example.com) into machine-readable IP addresses (e.g., 192.168.1.1). It is essential for locating devices and services on a network.	RFC 1034, RFC 1035	Application (7)

POP3	Used by email clients to retrieve emails from a mail server. It typically downloads emails to the client and removes them from the server.	RFC 1939	Application (7)
IMAP	Used for accessing and managing emails on a mail server. Unlike POP3, it allows users to organize emails in folders on the server and synchronize them across multiple devices.	RFC 3501	Application (7)
DHCP	Automatically assigns IP addresses and other network configuration parameters to devices on a network. It simplifies network administration by dynamically distributing network settings.	RFC 2131	Application (7)
TELNET	Provides a command-line interface for interacting with a remote computer or server over a network. It is a precursor to more secure protocols like SSH.	RFC 854	Application (7)
LPD	Enables users to send print jobs to a remote printer or print server over a network. It is a legacy protocol for network printing.	RFC 1179	Application (7)

3.2 Website Analysis

This section analyzes the network requests and responses involved in accessing the following websites:

- **canarabank.in**
- **github.com**
- **netflix.com**

The analysis was performed using the "Network" tab of the browser's developer tools (Chrome was used in this case).

3.2.1 canarabank.in

- **Request Line & Protocol Version:** Not applicable (no HTTP request was successfully made).
- **IP Address:** Could not be determined via DNS resolution. The DNS lookup failed, preventing the browser from establishing a connection to the server.
- **Persistent Connection:** Not applicable (no connection established).
- **HTTP Error Codes:** Not applicable (no HTTP communication occurred due to the prior DNS resolution error).
- **DNS Error:** `DNS_PROBE_FINISHED_NXDOMAIN` - This indicates a DNS resolution failure, meaning the browser could not find the IP address associated with the domain name `canarabank.in`. The domain might not exist or there could be an issue with the DNS server.

3.2.2 github.com

- **Request Line:** `GET / HTTP/2`
- **Protocol Version:** HTTP/2
- **IP Address:** `20.205.243.166` (obtained using the `ping` command in the terminal)
- **Persistent Connection:** Yes (HTTP/2 inherently supports persistent connections, which is further confirmed by the absence of a `Connection: close` header in the response.)
- **HTTP Error Codes:** `200 OK` (Indicates a successful request.)

Header Fields (github.com):

Request Headers:

- `:authority: github.com`
- `Sec-fetch-site: cross-site`
- `user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/132.0.0.0 Safari/537.36`

Response Headers:

- `date: Sat, 01 Feb 2025 06:23:46 GMT`

- `server: GitHub.com`
- `x-github-request-id: 8AA0:0A2C:961758:BDBE2A:679DBDF2`

Performance Metrics (github.com):

[Insert screenshot of performance metrics for github.com from browser developer tools]

(Note: Replace `[Insert screenshot of performance metrics for github.com...]` with your actual screenshot of the performance metrics. You can name the image `github_performance.png`.)

Cookies (github.com):

[Insert screenshot of cookies for github.com from browser developer tools]

(Note: Replace `[Insert screenshot of cookies for github.com...]` with your actual screenshot of the cookies. You can name the image `github_cookies.png`.)

3.2.3 netflix.com

- **Request Line:** `GET / HTTP/2` (Note: You mentioned "GET / HTTPS" which is not a standard request line. It is likely "GET / HTTP/2" as most sites use HTTP/2 for HTTPS traffic)
- **Protocol Version:** HTTP/2 (Assuming HTTPS was used, and HTTP/2 is common for secure connections)
- **IP Address:** `44.242.60.85` (obtained using the `ping` command)
- **Persistent Connection:** Yes (Likely persistent as HTTP/2 is used and there's no indication of connection closing in the response.)
- **HTTP Error Codes:** `302 Found` (This is a redirection status code. It indicates that the requested resource has been temporarily moved to a different URL. The browser will automatically follow the redirection to the new location provided in the `Location` header of the response.)

Header Fields (netflix.com):

Request Headers:

- `:authority: www.netflix.com`
- `sec-fetch-site: none`
- `user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/132.0.0.0 Safari/537.36`

Response Headers:

- `date: Sat, 01 Feb 2025 06:52:59 GMT`
- `server: envoy`
- `x-request-id: 627aac06-cc8f-466b-84a1-5f1df25c57ed`

Performance Metrics (netflix.com):

[Insert screenshot of performance metrics for netflix.com from browser developer tools]

(Note: Replace `[Insert screenshot of performance metrics for netflix.com...]` *with your actual screenshot. You can name the image* `netflix_performance.png`*.)*

Cookies (netflix.com):

[Insert screenshot of cookies for netflix.com from browser developer tools]

(Note: Replace `[Insert screenshot of cookies for netflix.com...]` *with your actual screenshot. You can name the image* `netflix_cookies.png`*.)*

3.3 HTTP Error Codes - General Overview

HTTP error responses are categorized into five classes:

- **1xx Informational Responses (100-199):** These indicate that the request was received and the server is continuing the process.
- **2xx Successful Responses (200-299):** These indicate that the request was successfully received, understood, and accepted.
- **3xx Redirection Responses (300-399):** These indicate that further action needs to be taken by the client to complete the request (e.g., following a redirection).
- **4xx Client Error Responses (400-499):** These indicate that the client seems to have made an error (e.g., requesting a resource that doesn't exist).
- **5xx Server Error Responses (500-599):** These indicate that the server failed to fulfill a valid request due to an error on the server's side.

Conclusion

This assignment provided practical experience in packet sniffing, network traffic analysis, and protocol investigation. We successfully implemented a

Python-based packet sniffer using `scapy` and analyzed network metrics from replayed pcap data. We also extended the sniffer to identify a hidden message within the traffic. Furthermore, using Wireshark and browser developer tools, we explored various application layer protocols and website loading characteristics, gaining insights into real-world network communication. The speed testing experiments highlighted the performance capabilities of our sniffer in different deployment scenarios.

This assignment deepened our understanding of network protocols, packet structure, and the tools used for network analysis, which are essential concepts in computer networks.