Assignment 1 Part 3

1 Network Traffic Analysis

This section analyzes a network traffic trace corresponding to a speed test using the M-Lab NDT7 tool. The NDT7 speed test works by flooding the network path between client and server for a pre-decided duration and logging the observed throughput in both downlink (server to client) and uplink (client to server) directions.

1.1 Objectives

The analysis script aims to achieve the following objectives:

1. Isolate the traffic corresponding to the speed test from background traffic and calculate the percentage of speed test traffic.
2. Plot a time series of observed throughput over time in each direction.
3. Find the average download and upload speeds.

1.2 Implementation

Our Python script, `speedtest_analysis.py`, uses the Scapy library to parse the PCAP file and perform the required analysis.

We have used argparse for command-line argument handling, allowing us to get different outputs from different flags. It requires a pcap file input and offers flags for extended analysis: `--plot` for time series visualization and `--throughput` for speed calculations. The core functionality includes a custom packet filtering mechanism (`isolate_traffic()` function) to identify speed test-related packets. We calculate the percentage of speed-test packets relative to the total captured packets. When specified, it can also generate time-series plots and calculate throughput, providing us a deeper understanding of network performance over time.

1.2.1 Isolating Speed Test Traffic

The script uses a custom filter function `isolate_traffic()` to identify packets related to the speed test:

```python
def isolate_traffic(pkt):
    if Ether in pkt and IP in pkt and TCP is in pkt:
        ether_layer = pkt[Ether]
        ip_layer = pkt[IP]
        tcp_layer = pkt[TCP]
        eth_condition = (
            ether_layer.src == "b8:81:98:9e:ba:01"
            and ether_layer.dst == "a8:da:0c:c5:b5:c3"
        ) or (
            ether_layer.dst == "b8:81:98:9e:ba:01"
            and ether_layer.src == "a8:da:0c:c5:b5:c3"
        )
        ip_condition = (
            ip_layer.src == "61.246.223.11" and ip_layer.dst == "192.168.29.159"
        ) or (ip_layer.dst == "61.246.223.11" and ip_layer.src == "192.168.29.159")
        tcp_condition = (
            (tcp_layer.dport == 46000 and tcp_layer.sport == 443)
            or (tcp_layer.sport == 46000 and tcp_layer.dport == 443)
            or (tcp_layer.dport == 39070 and tcp_layer.sport == 443)
            or (tcp_layer.sport == 39070 and tcp_layer.dport == 443)
        )
        return eth_condition, ip_condition and tcp_condition
```

```
    else:
        return False
```

This function filters packets based on specific Ethernet addresses, IP addresses, and TCP ports associated with the speed test. In our recent network communication analysis, we focused on the interaction between two specific MAC addresses: b8:81:98:9e:ba:01 and a8:da:0c:c5:b5:c3. Our investigation began at the link layer, where we filtered out the traffic between these two addresses to isolate their communication. Moving up to the network layer, we further refined our analysis by concentrating on traffic using the TCP protocol.

The analysis revealed that the NDT7 test, which was part of our examination, utilizes the TCP BBR protocol in its underlying code. A key aspect of TCP communication is its three-way handshake, which we observed occurring twice during our monitoring period. This handshake is a crucial process in TCP connections, consisting of three steps: the client sends a SYN packet, the server responds with a SYN-ACK packet, and finally, the client acknowledges an ACK packet. This sequence ensures a reliable connection is established before any data transfer begins.

Interestingly, we noted that most of the traffic we observed was channeled through a handshake performed on port 46000 for download and 39070 for upload. This concentration of activity on a specific port provides insights into the nature of the communication between the two MAC addresses. Additionally, we found that the TCP protocol was conducting network tests on port 443 on the server side. We developed a custom function to analyze network traffic, implementing a two-step filtering process. First, we filtered packets using the TCP protocol, followed by an IP protocol filter, allowing us to isolate specific network communications for detailed examination. Then we filtered the packets based on the port.

We identified that approximately **64.72%** of the network traffic was utilized in performing the test. This calculation was performed using our custom script. The result is automatically printed in the terminal upon executing the `isolate_traffic` function for any network file.

1.2.2 Plotting Time-Series of Throughput

Our function begins by iterating through a series of filtered network packets, each categorized as either download or upload traffic based on source and destination IP addresses with ports. We calculate each packet's data length in kilobytes and associate it with a specific second of the capture period. This meticulous categorization allows us to build a detailed, second-by-second profile of network activity, that is it allows to calculate the average network speed per second and plot it.

Following the initial data collection phase, our logic calculates the average throughput per second for download and upload speeds. This involves aggregating the accumulated data for each second of the capture period and computing the mean speeds. The resulting time series data clearly shows how network performance varied throughout the monitored time frame. The final output of our analysis presents a plot generated using Python's `matplotlib`, which gives us a picture of network performance, essential for understanding and improving the efficiency of data transfer processes. Note that the speed has been measured in Kbps, and we have taken the time stamp for the first packet in the filtered packets to be the reference for plotting

The `plot_time_series()` function generates time-series plots for both download and upload speeds:

```
def plot_time_series(filtered_packets):
    download_speeds = defaultdict(list)
    upload_speeds = defaultdict(list)
    times = []

    start_time = filtered_packets[0].time
    end_time = filtered_packets[-1].time

    for packet in filtered_packets:
        timestamp = packet.time
        second = int(timestamp - start_time)
        src_ip = packet[IP].src
```

```python
        dst_ip = packet[IP].dst
        buf = bytes(packet)
        data_len = len(buf) * 8 / 1e3

        if is_download(src_ip, dst_ip):
            download_speeds[second].append(data_len)
        elif is_upload(src_ip, dst_ip):
            upload_speeds[second].append(data_len)

    # Calculate average throughput per second
    avg_download_speeds = []
    avg_upload_speeds = []
    times = []

    for second in range(int(end_time - start_time) + 1):
        times.append(datetime.fromtimestamp(int(start_time + second)))
        avg_download_speeds.append(
            sum(download_speeds[second]) / max(1, len(download_speeds[second]))
        )
        avg_upload_speeds.append(
            sum(upload_speeds[second]) / max(1, len(upload_speeds[second]))
        )

    plt.figure(figsize=(10, 5))
    plt.fill_between(
        times,
        avg_download_speeds,
        alpha=0.5,
        label="Download Speed (Kbps)",
        color="#00BFFF",
    )
    # ... (plot configuration)
    plt.savefig("./time_series_download.png", dpi=1200, bbox_inches="tight")

    # Similar code for upload speed plot
```

This function calculates the average throughput per second and generates separate plots for download and upload speeds. The plots for the given pcap file are shown below :

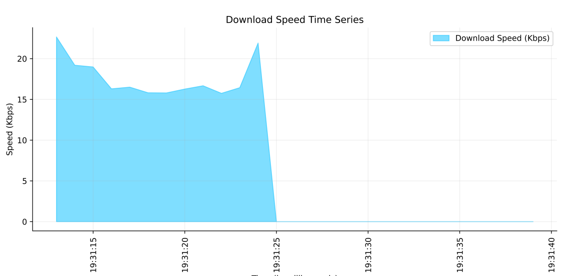

Figure 1: Download Speed

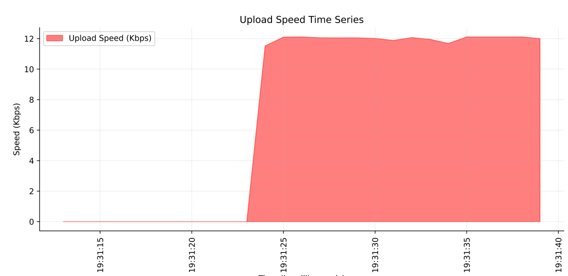

Figure 2: Upload Speed

Figure 3: Comparison of Download and Upload Speeds Over Time

1.2.3 Calculating Average Speeds

We implemented a crucial function called `calculate_speed` to determine the average download and upload speeds from filtered network packets. The function is provided with filtered traffic packets as the input. It begins by iterating through each packet, examining the source and destination IP addresses to categorize it as either download or upload traffic. For each packet, we convert its size to Megabits and accumulate these values separately for download and upload. This approach allows us to sum up the data transferred in each direction over the capture period. We use this information to calculate the download and upload time frame separately. We also store the download and upload packets in arrays to establish the time frame for the download and upload test.

Once all packets are processed, the function calculates the average speeds by dividing the total data transferred (in Megabits) by the capture duration (in seconds). This gives us the average speed of Megabits per second (Mbps) for download and upload. To ensure precision in our results, we round these values to three decimal places. The function then saves these results to a CSV file for further analysis and prints them to the console.

The `calculate_speed()` function computes the average download and upload speeds:

```python
def calculate_speed(filtered_packets):
    print("Calculating download and upload speed")

    if not filtered_packets:
        return 0, 0

    download = 0
    upload = 0
    download_packets = []
    upload_packets = []

    for packet in filtered_packets:
        src_ip = packet[IP].src
        dst_ip = packet[IP].dst
        buf = bytes(packet)
        data_len = len(buf) * 8 / 1e6  # Convert to Mbps
        src_port = packet[TCP].sport
        dst_port = packet[TCP].dport
        if is_download(src_ip, dst_ip, src_port, dst_port):
            download += data_len
            download_packets.append(packet)
        elif is_upload(src_ip, dst_ip, src_port, dst_port):
            upload += data_len
            upload_packets.append(packet)
    download_duration = (float)(download_packets[-1].time - download_packets[0].time)
    upload_duration = (float)(upload_packets[-1].time - upload_packets[0].time)

    if download_duration > 0:
        avg_download_speed = download / download_duration
    else:
        avg_download_speed = 0

    if upload_duration > 0:
        avg_upload_speed = upload / upload_duration
    else:
        avg_upload_speed = 0

    avg_download_speed = round(avg_download_speed, 2)
    avg_upload_speed = round(avg_upload_speed, 2)

    save_as_csv(
        avg_download_speed=avg_download_speed,
        avg_upload_speed=avg_upload_speed,
    )
    print(f"{avg_download_speed} Mbps", f"{avg_upload_speed} Mbps")
    return avg_download_speed, avg_upload_speed
```

This function calculates the total data transferred in each direction, divided by the test duration, and returns the average speeds in Mbps. The following is a sample output from the `calculate_speed` function, presented in CSV format rounded off up to 2 decimal places:

Download Speed (Mbps)	Upload Speed (Mbps)
29.38	8.48

1.3 Results

The script provides the following outputs:

1. Percentage of packets related to the speed test.
2. Time-series plots of download and upload speeds saved as PNG files.
3. Average download and upload speeds saved in a CSV file.

1.4 Usage

The script can be run from the command line with the following options:

```
python speedtest_analysis.py <pcap_file> [--plot] [--throughput]
```

Where:

- `<pcap_file>` is the path to the PCAP file to analyze.
- `--plot` generates the time-series plots.
- `--throughput` calculates and saves the average speeds.

1.5 Conclusion

This script provides a comprehensive analysis of the M-Lab NDT7 speed test traffic, isolating relevant packets, visualizing throughput over time, and calculating average speeds. The modular design allows for easy extension and modification for future analyses.