Aalto University
School of Electrical
Engineering

ELEC-E7130 Internet Traffic Measurements and Analysis

Final Assignment: From measurements to conclusions

Name: Xingji Chen

Student ID: 101659554

E-mail: xingji.chen@aalto.fi

Aalto University
School of Electrical
Engineering

Task 1: Capturing data

- Acquiring packet capture data

  - What kind of trace file and tool/s you are using to perform the packet capture.

    Wireshark.

  - Date, time, duration, measurement setting (in terms of profile if you are using the Wireshark) or file name if you are using the some public traces.

    Date, time, duration: 2023.11.20 from 19:00 to 20:00.
    Measurement setting: Default measurement setting of Wireshark.
    File name: final.pcap

Provide a short sample (10 lines or so) of the data taken from your capture file.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	Tp-LinkT_c3:e6:ed	IntelCor_32:9b:f0	ARP	42	Who has 192.168.1.110? Tell 192.168.1.1
2	0.000012	IntelCor_32:9b:f0	Tp-LinkT_c3:e6:ed	ARP	42	192.168.1.110 is at 50:e0:85:32:9b:f0
3	0.531969	192.168.1.103	239.255.255.250	SSDP	217	M-SEARCH * HTTP/1.1
4	0.840743	162.247.241.14	192.168.1.110	TCP	54	443 → 49819 [ACK] Seq=1 Ack=1 Win=8 Len=0
5	0.840779	192.168.1.110	162.247.241.14	TCP	54	[TCP ACKed unseen segment] 49819 → 443 [ACK] Seq=1 Ack=2 Win=1021 Len=0
6	0.881415	3.74.145.219	192.168.1.110	TCP	54	2099 → 63091 [ACK] Seq=1 Ack=1 Win=49 Len=0
7	0.881457	192.168.1.110	3.74.145.219	TCP	54	[TCP ACKed unseen segment] 63091 → 2099 [ACK] Seq=1 Ack=2 Win=1023 Len=0
8	1.146736	162.247.241.14	192.168.1.110	TCP	54	443 → 59650 [ACK] Seq=1 Ack=1 Win=8 Len=0
9	1.146770	192.168.1.110	162.247.241.14	TCP	54	[TCP ACKed unseen segment] 59650 → 443 [ACK] Seq=1 Ack=2 Win=1022 Len=0
10	2.679451	18.167.4.206	192.168.1.110	TCP	66	443 → 61171 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM WS=128

- Data pre-processing

  - Commands or code that is used in pre-processing for each case.

    Converting packet trace to flow data (PS2)

```
tshark -r finalr.pcap -q -z conv,tcp > final.txt
```

    TCP connection statistics (PS3)

```
tcptrace -l -r -n --csv final.pcap > finaltcp.csv
```

  - Short samples (10 lines or so) of the distilled data in each case (for PS3, one connection summary is enough).

Aalto University
School of Electrical
Engineering

PS1:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	Tp-LinkT_c3:e6:ed	IntelCor_32:9b:f0	ARP	42	Who has 192.168.1.110? Tell 192.168.1.1
2	0.000012	IntelCor_32:9b:f0	Tp-LinkT_c3:e6:ed	ARP	42	192.168.1.110 is at 50:e0:85:32:9b:f0
3	0.531969	192.168.1.103	239.255.255.250	SSDP	217	M-SEARCH * HTTP/1.1
4	0.840743	162.247.241.14	192.168.1.110	TCP	54	443 → 49819 [ACK] Seq=1 Ack=1 Win=8 Len=0
5	0.840779	192.168.1.110	162.247.241.14	TCP	54	[TCP ACKed unseen segment] 49819 → 443 [ACK] Seq=1 Ack=2 Win=1021 Len=0
6	0.881415	3.74.145.219	192.168.1.110	TCP	54	2099 → 63091 [ACK] Seq=1 Ack=1 Win=49 Len=0
7	0.881457	192.168.1.110	3.74.145.219	TCP	54	[TCP ACKed unseen segment] 63091 → 2099 [ACK] Seq=1 Ack=2 Win=1023 Len=0
8	1.146736	162.247.241.14	192.168.1.110	TCP	54	443 → 59650 [ACK] Seq=1 Ack=1 Win=8 Len=0
9	1.146770	192.168.1.110	162.247.241.14	TCP	54	[TCP ACKed unseen segment] 59650 → 443 [ACK] Seq=1 Ack=2 Win=1022 Len=0
10	2.679451	18.167.4.206	192.168.1.110	TCP	66	443 → 61171 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM WS=128

PS2:

		<-		->		Total		Relative	Duration
		Frames	Bytes	Frames	Bytes	Frames	Bytes	Start	
192.168.1.110:61784	<-> 23.72.90.132:443	102140	153293101	43637	2722088	145777	156015189	2239.638624000	1784.1933
192.168.1.110:61788	<-> 23.72.90.132:443	67087	100566224	28512	1827766	95599	102393990	2283.368245000	1740.4608
192.168.1.110:61743	<-> 199.232.46.250:443	22789	33434593	11558	663818	34347	34098411	1752.930316000	2538.4990
192.168.1.110:61494	<-> 199.232.46.250:443	16736	24606693	8580	497592	25316	25104285	171.991906000	1517.4971
192.168.1.110:61634	<-> 23.72.90.132:443	3986	5932407	1892	122891	5878	6055298	746.653338000	1237.1683
192.168.1.110:61322	<-> 23.72.90.143:443	4061	6066636	1794	125224	5855	6191860	91.868856000	556.9600
192.168.1.110:61217	<-> 164.52.39.44:443	2695	1455516	2249	447162	4944	1902678	40.527595000	5352.2058
192.168.1.110:61330	<-> 101.91.133.30:443	1983	241500	1892	480073	3875	721573	97.318691000	2139.5346
192.168.1.110:61430	<-> 34.36.232.77:443	2630	3126997	1119	71153	3749	3198150	167.879253000	5224.3517
192.168.1.110:49819	<-> 162.247.241.14:443	1718	95064	1718	92963	3436	188027	0.840743000	8587.3059

PS3:

conn_#	host_a	host_b	port_a	port_b	first_packe	last_packet	total_packe	total_packe	resets_sent	resets_sent	ack_pkts_s	ack_pkts_s	pure_acks_	pure_acks_	sack_pkts_	sack_pkts_	dsack_pkts	dsack_pkts	max_sack_	max_sack_	unique_by	unique_by	actual_da
1	162.247.24	192.168.1.	443	49819	1.7E+09	1.7E+09	1718	1718	0	0	1718	1718	1718	1527	191	0	191	0	1	0	0	1	0
2	3.74.145.2	192.168.1.	2099	63091	1.7E+09	1.7E+09	513	573	0	0	513	573	414	499	0	0	0	0	0	0	51365	25507	99
3	162.247.24	192.168.1.	443	59650	1.7E+09	1.7E+09	1718	1718	0	0	1718	1718	1718	1527	191	0	191	0	1	0	0	1	0
4	18.167.4.2	192.168.1.	443	61171	1.7E+09	1.7E+09	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
5	18.167.4.2	192.168.1.	443	61172	1.7E+09	1.7E+09	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
6	192.168.1.	67.207.82.	61177	443	1.7E+09	1.7E+09	2	1	1	0	2	1	0	1	0	0	0	0	0	0	0	0	0
7	192.168.1.	18.167.4.2	61175	443	1.7E+09	1.7E+09	2	1	1	0	2	1	0	1	0	0	0	0	0	0	0	0	0
8	192.168.1.	18.167.4.2	61176	443	1.7E+09	1.7E+09	2	1	1	0	2	1	0	1	0	0	0	0	0	0	0	0	0
9	192.168.1.	23.15.241.	61181	443	1.7E+09	1.7E+09	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0
10	192.168.1.	20.37.241.	61184	443	1.7E+09	1.7E+09	12	13	0	0	11	13	5	4	0	2	0	2	0	1	1211	6606	9

- Packet data PS1

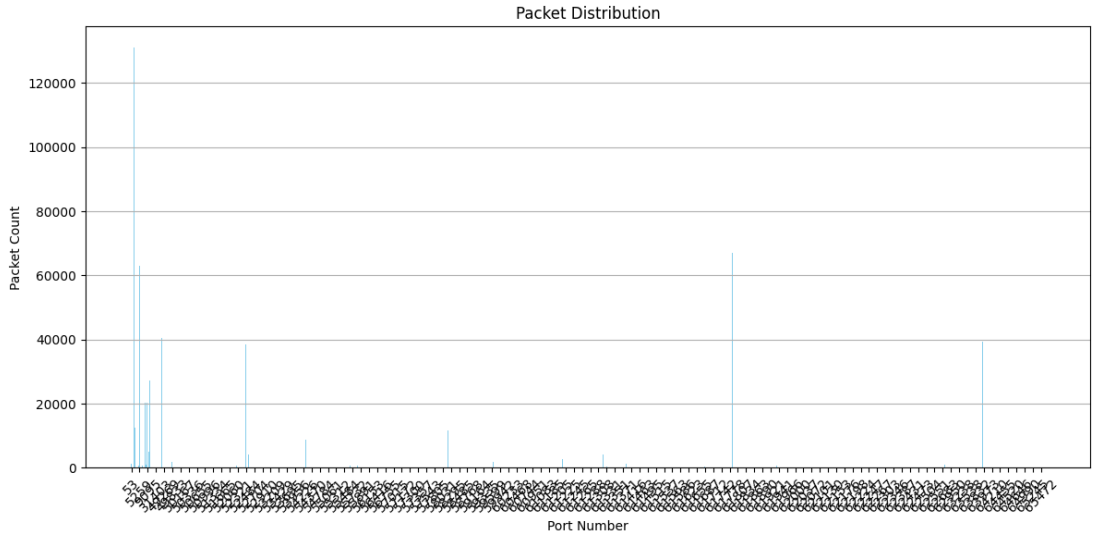
1. Visualize packet distribution by port numbers.

```
import pyshark
import matplotlib.pyplot as plt
from collections import Counter


def capture_packets(pcap_path, tshark_path):
    """
    Captures packets from a pcap file and returns a pyshark
FileCapture object.
    """
    return pyshark.FileCapture(pcap_path, tshark_path=tshark_path,
keep_packets=True)


def count_port_usage(packet_capture):
    """
```

```
    Counts the usage frequency of destination ports in the captured
packets.
    """
    port_counter = Counter()
    for packet in packet_capture:
        if 'TCP' in packet or 'UDP' in packet:
            transport_layer = packet.tcp if 'TCP' in packet else
packet.udp
            port_counter[transport_layer.dstport] += 1
    return port_counter


def plot_port_distribution(ports, counts):
    """
    Plots a bar chart showing the distribution of packet counts per
port number.
    """
    plt.figure(figsize=(12, 6))
    plt.bar(ports, counts, color='skyblue')
    plt.xlabel('Port Number')
    plt.ylabel('Packet Count')
    plt.title('Packet Distribution Across Ports')
    plt.grid(axis='y')

    # Display every nth label to avoid clutter
    label_step = 20
    plt.xticks([port for i, port in enumerate(ports) if i %
label_step == 0], rotation=45)
    plt.tight_layout()
    plt.show()


# Paths to the tshark executable and pcap file
tshark_executable = 'D:\\wireshark\\tshark.exe'
pcap_file = 'files/final.pcap'

# Capture and count port usage
packet_capture = capture_packets(pcap_file, tshark_executable)
port_usage = count_port_usage(packet_capture)
packet_capture.close()
```

```
# Prepare data for plotting
sorted_ports, packet_counts = zip(*sorted(port_usage.items(),
key=lambda x: int(x[0])))

# Plot distribution of packets across ports
plot_port_distribution(sorted_ports, packet_counts)
```

This code is designed to analyze network traffic in a pcap (packet capture) file, with a particular focus on the distribution of traffic across different network ports. It first reads and parses the pcap file, identifying the destination ports of TCP and UDP packets. Then, the code calculates the number of packets for each port. After processing, it visualizes this data in the form of a bar chart, displaying the packet count for each port.



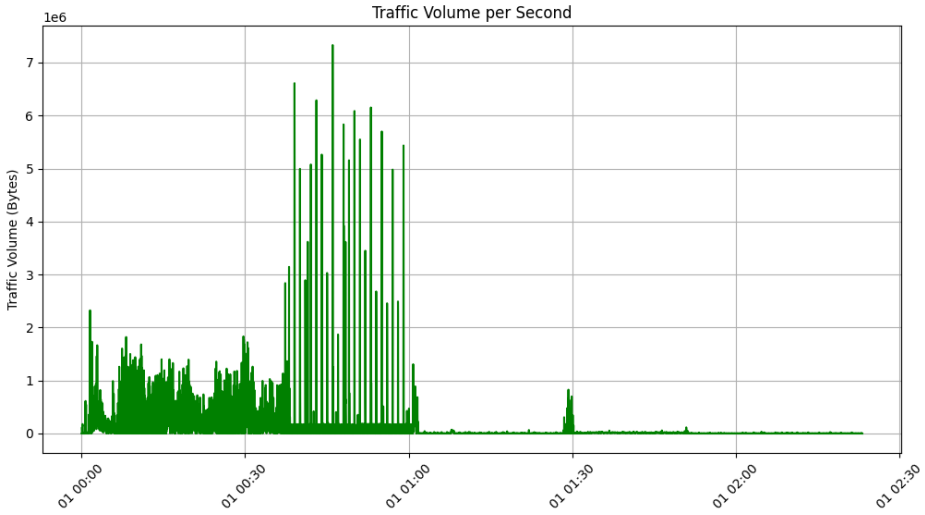
The chart exhibits several ports with notably high traffic, and these peaks likely correspond to commonly used, well-known ports typically employed for popular services or protocols. The packet count for the majority of port numbers is very low. This indicates that only a few ports are heavily utilized within this network. The distribution shows that network traffic is concentrated on common ports, which is typical in most network environments where certain services (such as HTTP, HTTPS, FTP, SMTP, etc.) dominate the traffic.

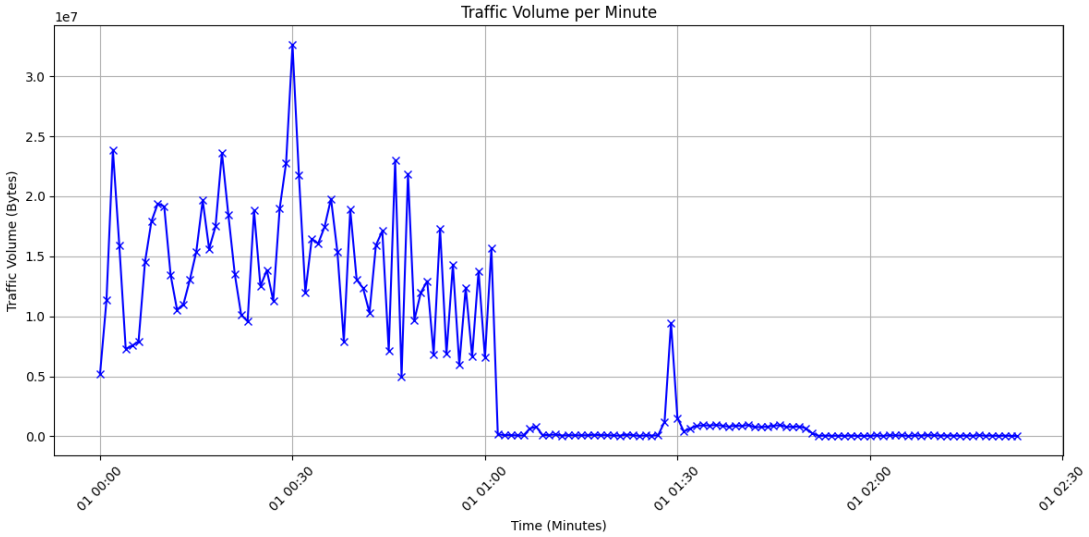
2. Plot traffic volume as a function of time with at least two sufficiently different time scales.

```
import pandas as pd
import matplotlib.pyplot as plt

def read_and_process_traffic_data(file_path):
    """
    Reads traffic data from a CSV file and converts the 'Time' column
to datetime.
    """
    data = pd.read_csv(file_path)
    data['Time'] = pd.to_datetime(data['Time'], unit='s')
    return data

def aggregate_traffic(data, resample_period):
    """
    Aggregates traffic data by summing the 'Length' column for the
given resampling period.
    """
    return
data.groupby('Time')['Length'].sum().reset_index().set_index('Time').
resample(resample_period).sum().fillna(0)

def plot_traffic(traffic, title, color, marker):
    """
    Plots the traffic volume as a function of time.
    """
    plt.figure(figsize=(10, 6))
    plt.plot(traffic.index, traffic['Length'], color=color,
marker=marker)
    plt.grid(True)
    plt.title(f'Traffic Volume per {title}')
    plt.xlabel(f'Time ({title})')
    plt.ylabel('Traffic Volume (Bytes)')
    plt.xticks(rotation=45)
    plt.tight_layout()

csv_file_path = 'files/final.csv'

traffic_data = read_and_process_traffic_data(csv_file_path)
```

```
traffic_data_by_second = aggregate_traffic(traffic_data, 'S')
traffic_data_by_minute = aggregate_traffic(traffic_data, 'T')

plot_traffic(traffic_data_by_second, 'Second', 'green', 'o')
plot_traffic(traffic_data_by_minute, 'Minute', 'blue', 'x')

plt.show()
```

This code reads a CSV file that contains timestamps and lengths of data packets. It then resamples this data by second and by minute to plot graphs showing how the traffic volume changes over time.





The graph with a second time scale shows a very detailed picture of traffic fluctuations. It is evident that at certain specific moments, the traffic reaches very high peaks. These peaks might indicate sudden surges in traffic, potentially caused by bursts of data packets or concentrated network activities. The volatility of the traffic is quite intense, suggesting that the network may face congestion and instability at these points in time.

When presented with a minute time scale, the same data is shown in a much smoother view. At this scale, the instantaneous peaks in traffic are less pronounced, but the patterns of fluctuation are still identifiable. The appearance of high traffic peaks at intervals suggests that the increase in traffic might have a periodic characteristic.

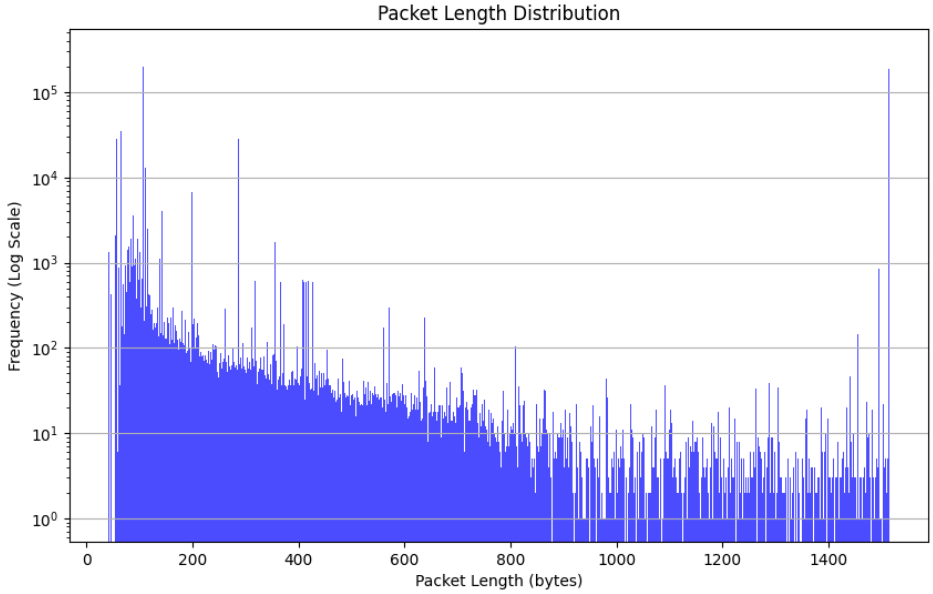
3. Plot packet length distribution (use bins of width 1 byte), its empirical cumulative distribution function and key summary statistics.

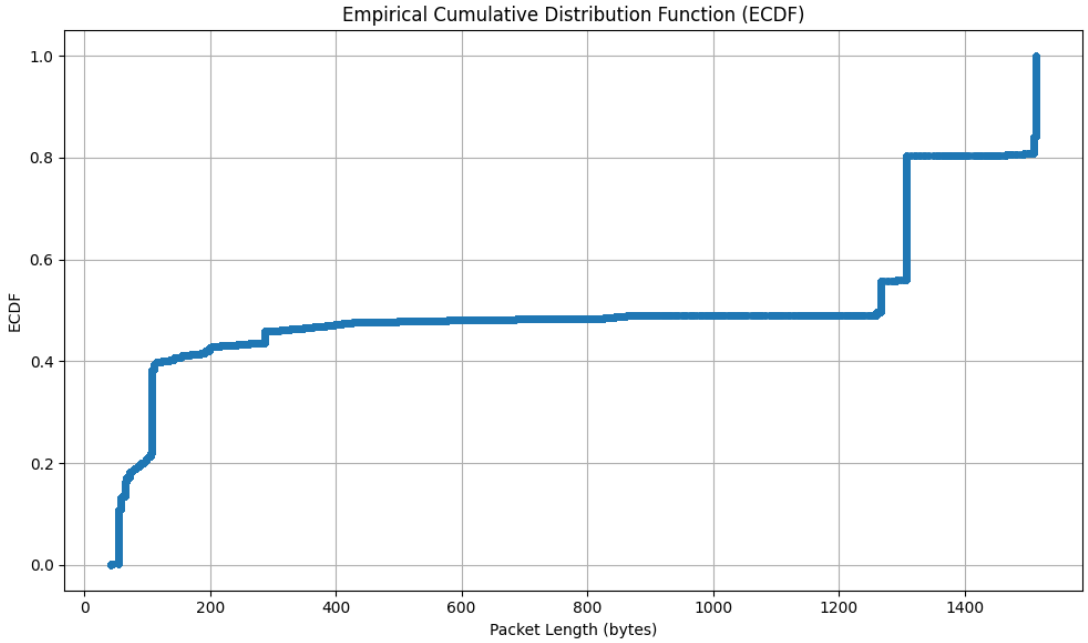
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

def load_data(file_path):
    return pd.read_csv(file_path)

def plot_histogram(data, column, bin_size):
    plt.figure(figsize=(10, 6))
    bins_range = range(min(data[column]), max(data[column]) +
bin_size, bin_size)
    plt.hist(data[column], bins=bins_range, color='blue', alpha=0.7,
log=True)
    plt.grid(axis='y')
    plt.title('Packet Length Distribution')
    plt.xlabel('Packet Length (bytes)')
    plt.ylabel('Frequency (Log Scale)')
    plt.show()

def plot_ecdf(data, column):
    plt.figure(figsize=(10, 6))
    sorted_data = np.sort(data[column])
    y_values = np.arange(1, len(sorted_data) + 1) / len(sorted_data)
    plt.plot(sorted_data, y_values, marker='.', linestyle='none',
```

```
color='blue')
    plt.grid(True)
    plt.title('Empirical Cumulative Distribution Function (ECDF)')
    plt.xlabel('Packet Length (bytes)')
    plt.ylabel('ECDF')
    plt.tight_layout()
    plt.show()

def print_summary_statistics(data, column):
    print(f"Summary statistics for {column}:")
    print(data[column].describe())

# Main code
csv_file_path = 'files/final.csv'
packet_data = load_data(csv_file_path)

# Plot the histogram of packet lengths
plot_histogram(packet_data, 'Length', bin_width=1)

# Plot the ECDF of packet lengths
plot_ecdf(packet_data, 'Length')

# Print summary statistics for packet lengths
print_summary_statistics(packet_data, 'Length')
```

This code is primarily used for analyzing and visualizing the distribution of network packet lengths. It achieves this by presenting the data in different aspects through a histogram and an Empirical Cumulative Distribution Function (ECDF) plot. Additionally, it provides some basic descriptive statistics about the data.

```
Summary statistics for packet lengths:
count    1.199511e+06
mean     7.663187e+02
std      6.365886e+02
min      4.200000e+01
25%      1.070000e+02
50%      1.267000e+03
75%      1.307000e+03
max      1.514000e+03
```





The first graph presents the distribution of packet lengths, allowing us to observe the spread of packet sizes in the measured network traffic. The logarithmic scale highlights that certain packet lengths are more common than others, with noticeable peaks at specific lengths.

The second graph illustrates the empirical cumulative distribution function (ECDF) of packet

lengths. This function provides a cumulative total that indicates the proportion of packets that are less than a certain length. The ECDF shows sharp increases at certain points, corresponding with the peaks observed in the packet length distribution graph.

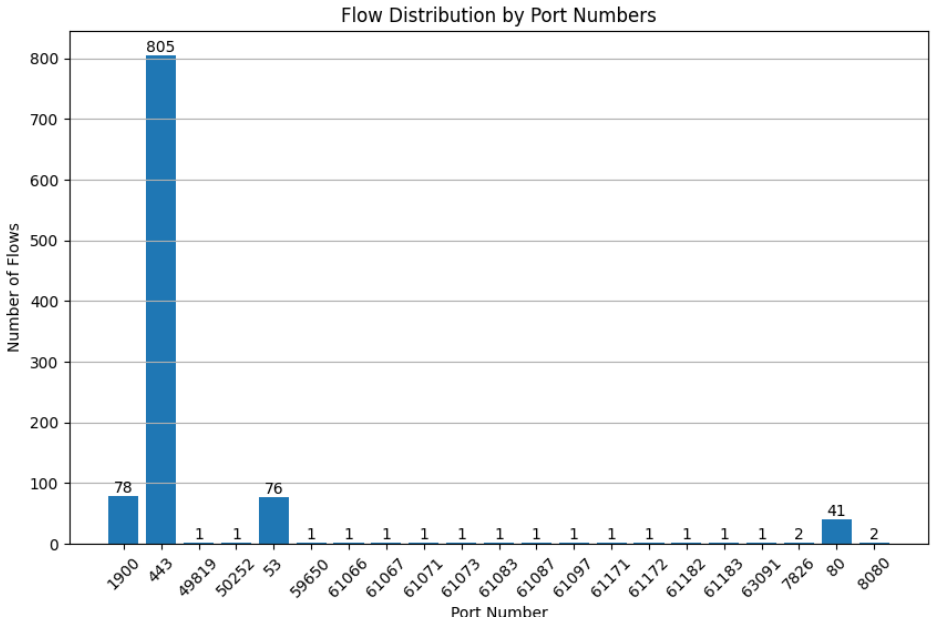
- Flow data PS2

4. Visualize flow distribution by port numbers.

```
import pandas as pd
import matplotlib.pyplot as plt

def convert_to_bytes(row):
    units = {'bytes': 1, 'kb': 1024, 'mb': 1024**2}

    try:
        load_bytes_unit = str(row.get('LD_Bytes_Unit',
'bytes')).lower()
        load_factor = units[load_bytes_unit]
        load_bytes = row.get('LD_Bytes', 0) * load_factor

        receive_bytes_unit = str(row.get('RD_Bytes_Unit',
'bytes')).lower()
        receive_factor = units[receive_bytes_unit]
        receive_bytes = row.get('RD_Bytes', 0) * receive_factor

        total_bytes_unit = str(row.get('Total_Bytes_Unit',
'bytes')).lower()
        total_factor = units[total_bytes_unit]
        total_bytes = row.get('Total_Bytes', 0) * total_factor

        return pd.Series({
            'Load_Bytes': load_bytes,
            'Receive_Bytes': receive_bytes,
            'Total_Bytes': total_bytes,
            'Server_IP': row.get('Destination_IP', '')
        })

    except KeyError as e:
        print(f"Error processing row {row}: {e}")
        raise ValueError("Invalid unit. Supported units are 'bytes',
```

```
'kb', 'mb.")

# Reading and preparing data
data_file = 'files/final.txt'
data_columns = ["Source_IP", "Arrow", "Destination_IP", "LD_Frames",
"LD_Bytes", "LD_Bytes_Unit",
             "RD_Frames", "RD_Bytes", "RD_Bytes_Unit",
"Total_Frames", "Total_Bytes", "Total_Bytes_Unit",
             "Start", "Duration"]

df = pd.read_csv(data_file, sep='\s+', skiprows=5, header=None,
skipfooter=1, engine='python')
df.columns = data_columns
df = df.assign(**df.apply(convert_to_bytes, axis=1))
df['Port'] = df['Destination_IP'].str.split(':').str[1].astype(str)

port_flow_distribution = df.groupby('Port').size()

plt.figure(figsize=(10, 6))
bar_chart = plt.bar(port_flow_distribution.index,
port_flow_distribution.values)

for bar in bar_chart:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width() / 2, height, int(height),
va='bottom', ha='center')

plt.xlabel('Port Number')
plt.ylabel('Number of Flows')
plt.title('Flow Distribution by Port Numbers')
plt.xticks(rotation=45)
plt.grid(axis='y')
plt.show()
```

This code is primarily used for analyzing and visualizing network traffic data. Initially, it reads data from a file and converts it into an appropriate format, such as converting the size of data packets from various units into bytes. Then, the code groups this data by port number and calculates the amount of traffic on each port. Finally, it uses the matplotlib library to draw a bar chart, displaying the distribution of traffic across different ports.



This chart illustrates the distribution of network flows categorized by port numbers. It is evident that the traffic on certain ports is significantly higher than on others. Most notably, port 443 shows over 800 flows, which is commonly used for secure web transmission (HTTPS), suggesting a substantial amount of secure web traffic. Port 80 also displays a higher volume, with about 41 flows, typically associated with HTTP service, indicative of unencrypted web traffic. Other ports such as 22, usually used for SSH, and 3389, for Remote Desktop Protocol, have comparatively lower traffic, yet their presence indicates remote administration activities on the network. The majority of other ports have only 1 or 2 flows, pointing to specific, non-standard services or sporadic network communications.

5. Plot traffic volume as a function of time with at least two sufficiently different time scales.

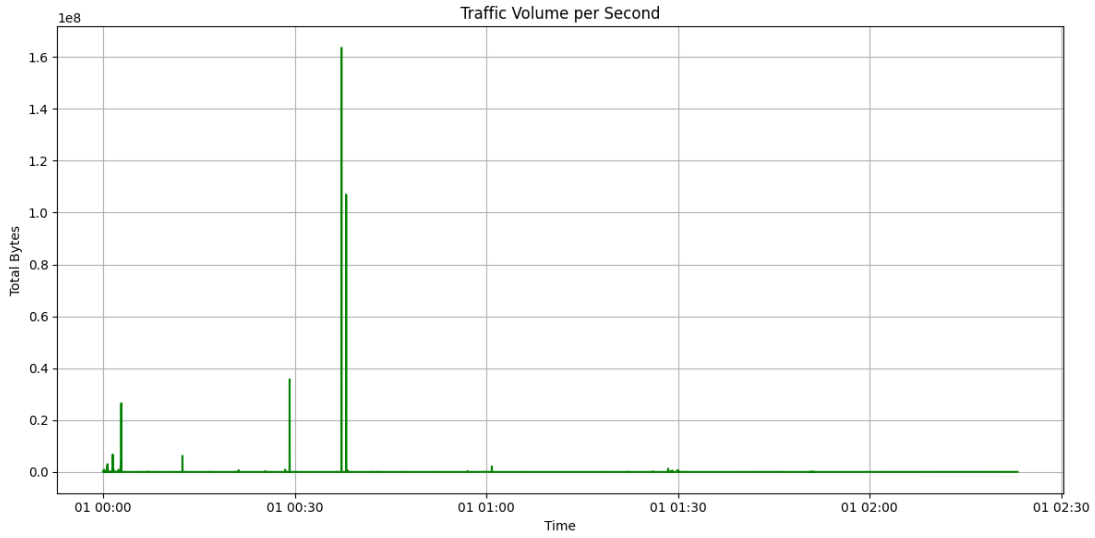
```
import pandas as pd
import matplotlib.pyplot as plt
import re
import datetime

def parse_line(line):
    match =
re.search(r'(\d+\.\d+\.\d+\.\d+:\d+)\s+<->\s+(\d+\.\d+\.\d+\.\d+:\d+)
\s+(\d+)\s+(\d+\s+\w+)\s+(\d+)\s+(\d+\s+\w+)\s+(\d+)\s+(\d+\s+\w+)\s+
(\d+\.\d+)', line)
```

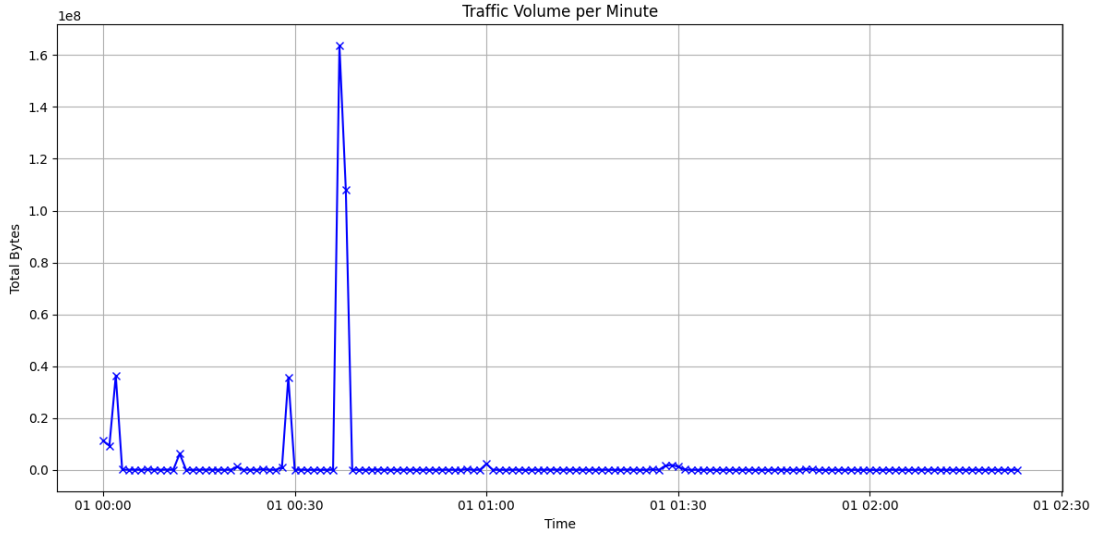
```
    if match:
        return {
            "Source_IP": match.group(1),
            "Destination_IP": match.group(2),
            "Upload_Frames": int(match.group(3)),
            "Upload_Bytes": match.group(4),
            "Download_Frames": int(match.group(5)),
            "Download_Bytes": match.group(6),
            "Total_Frames": int(match.group(7)),
            "Total_Bytes": match.group(8),
            "Start": float(match.group(9))
        }
    else:
        return None

# Function to convert the total bytes to a uniform unit (bytes)
def convert_bytes(byte_str):
    number, unit = byte_str.split()
    number = float(number)
    unit = unit.lower()
    if unit == 'kb':
        return number * 1024
    elif unit == 'mb':
        return number * 1024 * 1024
    elif unit == 'gb':
        return number * 1024 * 1024 * 1024
    else:
        return number

# Parsing the entire file
file_path = 'files/final.txt'  # Replace with your file path
parsed_full_data = []
with open(file_path, 'r') as file:
    for _ in range(5):  # Skipping the first five lines
        next(file)
    for line in file:  # Parsing each line in the file
        parsed_line = parse_line(line)
        if parsed_line:
            parsed_line["Total_Bytes"] =
convert_bytes(parsed_line["Total_Bytes"])
```

```
            parsed_full_data.append(parsed_line)

# Convert the parsed data to a DataFrame
full_df = pd.DataFrame(parsed_full_data)

# Convert the 'Start' column to a datetime format using a reference
date
reference_date = datetime.datetime(1970, 1, 1)
full_df['Start'] = pd.to_datetime(full_df['Start'], unit='s',
origin=reference_date)

# Resampling data to seconds
df_seconds = full_df.resample('1S', on='Start').sum()

# Plotting the data for second-wise
plt.figure(figsize=(12, 6))
plt.plot(df_seconds.index, df_seconds['Total_Bytes'], color='green',
marker='o')
plt.title('Traffic Volume per Second')
plt.xlabel('Time')
plt.ylabel('Total Bytes')
plt.grid(True)
plt.tight_layout()
plt.show()

df_minutes = full_df.resample('1T', on='Start').sum()

plt.figure(figsize=(12, 6))
plt.plot(df_minutes.index, df_minutes['Total_Bytes'], color='blue',
marker='x')
plt.title('Traffic Volume per Minute')
plt.xlabel('Time')
plt.ylabel('Total Bytes')
plt.grid(True)
plt.tight_layout()
plt.show()
```

This code functions to read network traffic data from a CSV file and convert it into a time series format. Subsequently, it groups the data by timestamps and calculates the total traffic for each time period, measured in seconds and minutes. Finally, the code uses the matplotlib library to plot the

network traffic variations per second and per minute, respectively.





In the first graph, we observe a dramatic increase in traffic volume at specific seconds, indicating moments of high activity or data transfer. The most significant peak is close to 1.6e8 bytes, suggesting a momentary but substantial surge in traffic. Aside from these peaks, the pattern shows relatively low traffic volume, indicating that such high volumes are not sustained over long periods.

The second graph provides a minute-by-minute aggregation of traffic volume. Here, peaks are less pronounced due to the aggregation over each minute, but we still observe peaks at specific intervals,

including one that corresponds to the highest peak seen in the per-second graph. This minute-scale view smooths out the extreme fluctuations seen on a per-second basis, yet still highlights moments of intense activity.

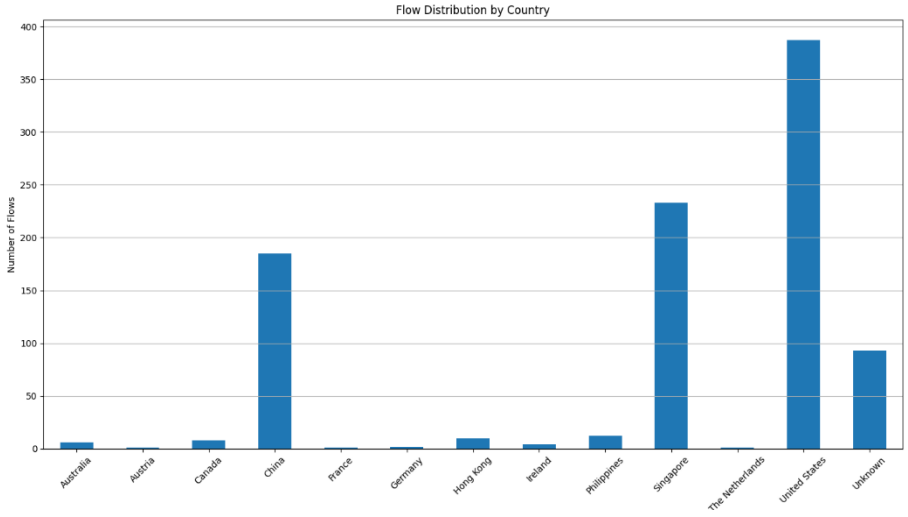
6. Visualize flow distribution by country.

```
import pandas as pd
import matplotlib.pyplot as plt
from geoip2.database import Reader

def convert_bytes(row):
    units = {'bytes': 1, 'kb': 1024, 'mb': 1024**2}

    try:
        upload_unit = str(row['Upload_Bytes_Unit']).lower()
        upload_factor = units[upload_unit]
        upload_bytes = row['Upload_Bytes'] * upload_factor

        download_unit = str(row['Download_Bytes_Unit']).lower()
        download_factor = units[download_unit]
        download_bytes = row['Download_Bytes'] * download_factor

        total_unit = str(row['Total_Bytes_Unit']).lower()
        total_factor = units[total_unit]
        total_bytes = row['Total_Bytes'] * total_factor

        return pd.Series({
            'Upload_Bytes': upload_bytes,
            'Download_Bytes': download_bytes,
            'Total_Bytes': total_bytes,
            'Server_IP': row['Destination_IP']
        })
    except KeyError as e:
        print(f"Error processing row {row}: {e}")
        raise ValueError("Invalid unit. Supported units are 'bytes',
'kb', 'mb.")

data_file = 'files/final.txt'
column_names = [
```

```
    "Source_IP", "Arrow", "Destination_IP", "Upload_Frames",
"Upload_Bytes", "Upload_Bytes_Unit",
    "Download_Frames", "Download_Bytes", "Download_Bytes_Unit",
"Total_Frames", "Total_Bytes", "Total_Bytes_Unit",
    "Start", "Duration"
]

traffic_data = pd.read_csv(data_file, sep='\s+', skiprows=5,
header=None, skipfooter=1, engine='python')
traffic_data.columns = column_names
traffic_data =
traffic_data.assign(**traffic_data.apply(convert_bytes, axis=1))
geoip_file = 'others/GeoLite2-Country.mmdb'
geoip_reader = Reader(geoip_file)

def get_country(ip):
    try:
        response = geoip_reader.country(ip)
        return response.country.name
    except:
        return "Unknown"

traffic_data['Country'] =
traffic_data['Destination_IP'].str.split(':').str[0].apply(get_countr
y)

country_traffic_distribution = traffic_data.groupby('Country').size()
plt.figure(figsize=(11, 7.5))
country_traffic_distribution.plot(kind='bar')
plt.grid(axis='y')
plt.xlabel('Country')
plt.ylabel('Number of Flows')
plt.title('Flow Distribution by Country')
plt.xticks(rotation=45)
plt.show()
```

This code is primarily focused on analyzing and visualizing network traffic data, with a specific emphasis on the geographical distribution of traffic. It reads network traffic data from a text file, including the size of data packets and their destination IP addresses. The code then uniformly

converts the size of data packets from various units into bytes and uses a GeoIP database to map each IP address to its corresponding country. Finally, the code uses the matplotlib library to create a bar chart, displaying the distribution of network traffic across different countries.



The bar chart visualizes the distribution of network flows by country. The United States has the highest number of flows, followed by the Netherlands and China, indicating a higher level of internet traffic activity or more network interactions originating from or destined to these countries.

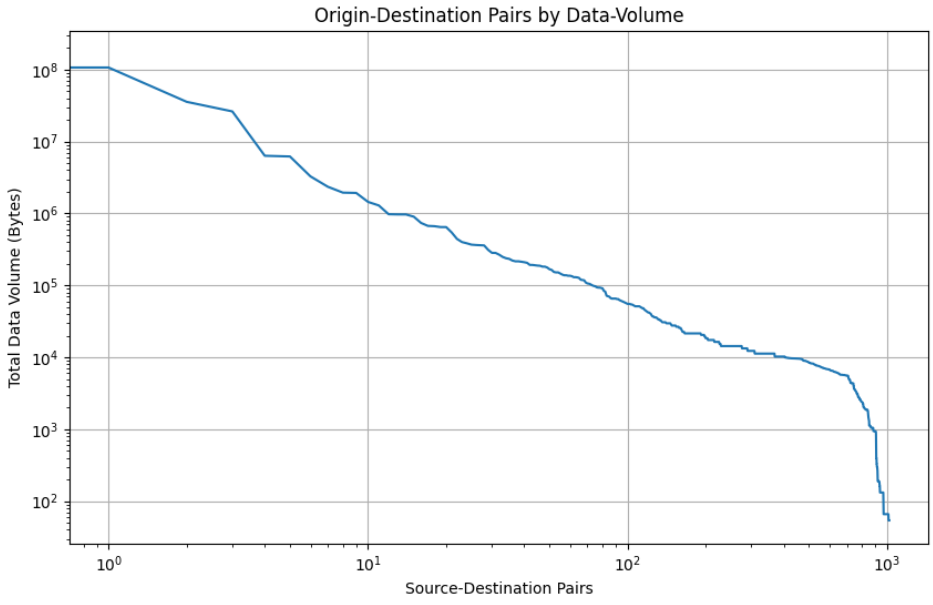
7. Plot origin-destination pairs both by data volume and by flows (Zipf type plot).

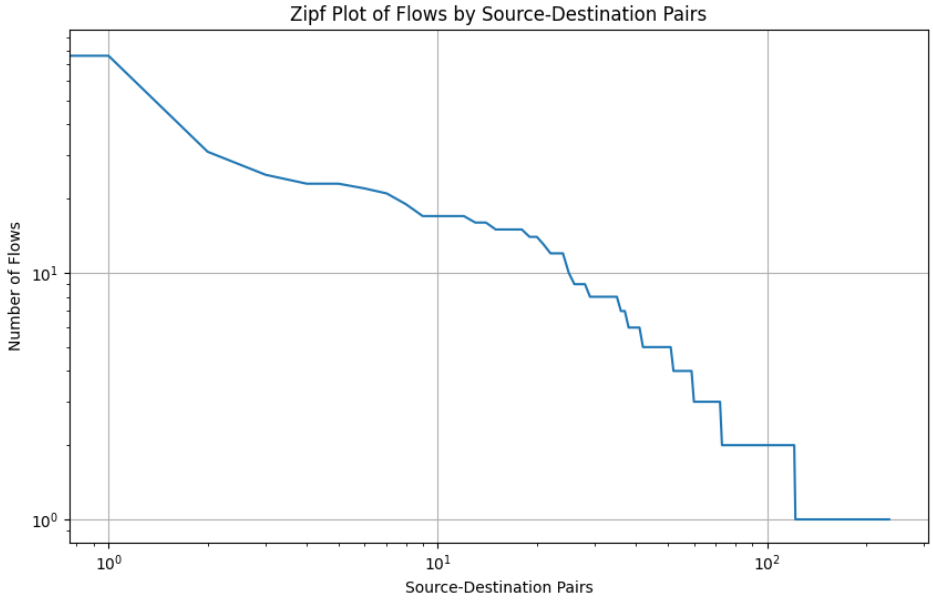
```
import pandas as pd
import matplotlib.pyplot as plt

def convert_units_to_bytes(row):
    units = {'bytes': 1, 'kb': 1024, 'mb': 1024**2}

    try:
        upload_unit = str(row['Upload_Unit']).lower()
        upload_factor = units[upload_unit]
        upload_bytes = row['Upload_Bytes'] * upload_factor

        download_unit = str(row['Download_Unit']).lower()
        download_factor = units[download_unit]
        download_bytes = row['Download_Bytes'] * download_factor

        total_unit = str(row['Total_Unit']).lower()
        total_factor = units[total_unit]
```

```
        total_bytes = row['Total_Bytes'] * total_factor

        return pd.Series({
            'Upload_Bytes': upload_bytes,
            'Download_Bytes': download_bytes,
            'Total_Bytes': total_bytes,
            'Destination_IP': row['Destination_IP']
        })
    except KeyError as e:
        print(f"Error processing row {row}: {e}")
        raise ValueError("Invalid unit. Supported units are 'bytes',
'kb', 'mb.")

data_file = 'files/final.txt'
column_names = [
    "Source_IP", "Arrow", "Destination_IP", "Upload_Frames",
"Upload_Bytes", "Upload_Unit",
    "Download_Frames", "Download_Bytes", "Download_Unit",
"Total_Frames", "Total_Bytes", "Total_Unit",
    "Start_Time", "Duration"]

network_data = pd.read_csv(data_file, sep='\s+', skiprows=5,
header=None, skipfooter=1, engine='python')
network_data.columns = column_names
network_data =
network_data.assign(**network_data.apply(convert_units_to_bytes,
axis=1))

network_data['Src_Dst_Pair'] = network_data['Source_IP'] + ' - ' +
network_data['Destination_IP']

pairwise_traffic_data =
network_data.groupby('Src_Dst_Pair')['Total_Bytes'].sum()
pairwise_flow_counts = network_data.groupby('Src_Dst_Pair').size()
sorted_traffic = pairwise_traffic_data.sort_values(ascending=False)
sorted_flows = pairwise_flow_counts.sort_values(ascending=False)

plt.figure(figsize=(10, 6))
plt.plot(sorted_traffic.values)
plt.grid(True)
```

```
plt.xlabel('Origin-Destination Pairs')
plt.ylabel('Data Volume (Bytes)')
plt.title('Origin-destination pairs by data volume')
plt.yscale('log')
plt.xscale('log')
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(sorted_flows.values)
plt.grid(True)
plt.xlabel('Origin-Destination Pairs')
plt.ylabel('Flow Counts')
plt.title('Origin-destination pairs by flows')
plt.yscale('log')
plt.xscale('log')
plt.show()
```

This code functions to read network traffic data from a text file and processes it for analysis. Initially, it converts the size of data packets from various units into bytes and associates each packet with its corresponding source-destination IP pair. The code then calculates the total data volume and the number of flows for each source-destination pair, sorts this data, and plots two distributions: one showing the data volume distribution sorted by source-destination pairs, and another depicting the distribution of the number of flows.





The first graph displays the distribution of network traffic among origin-destination pairs by data volume, where a small number of pairs account for a large portion of the total data volume, indicating that the majority of data transmission in the network is concentrated among a few pairs. The second graph, plotted by the number of flows, shows a distribution that also follows a Zipf-like pattern. This demonstrates that a few origin-destination pairs have a high number of flows, while the majority of pairs have relatively few. This is a typical characteristic of networks where a small number of hosts or services manage most connections.

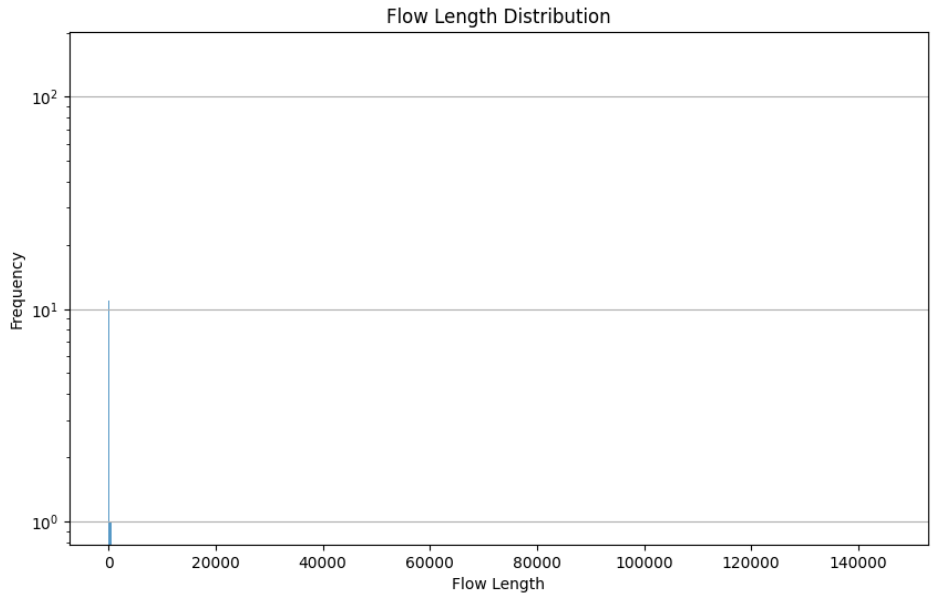
8. Plot flow length distribution, its empirical cumulative distribution function and key summary statistics.

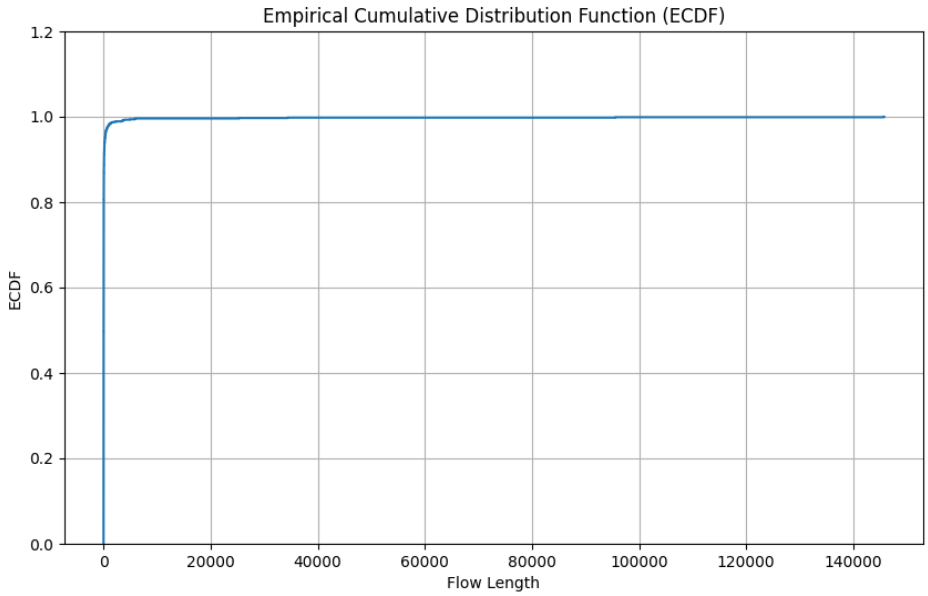
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

def convert_to_kilobytes(row):
    units = {'bytes': 1, 'kb': 1024, 'mb': 1024 ** 2}
    converted_data = {}

    for column_prefix in ['ld', 'rd', 'total']:
        bytes_value = row[f'{column_prefix}_bytes']
        unit = str(row[f'{column_prefix}_bytes_unit']).lower()
```

```
        try:
            factor = units[unit]
        except KeyError:
            raise ValueError(
                f"Invalid unit '{unit}' for {column_prefix}_bytes.
Supported units are 'bytes', 'kb', 'mb'.")

        converted_data[f'{column_prefix}_kilobytes'] = bytes_value *
factor

    return pd.Series({**converted_data, 'server_ip':
row['second_ip']})


# Read and process data
data_file = 'files/final.txt'
data_columns = ["first_ip", "->", "second_ip", "ld_frames",
"ld_bytes", "ld_bytes_unit",
                "rd_frames", "rd_bytes", "rd_bytes_unit",
"total_frames", "total_bytes", "total_bytes_unit",
                "start_time", "duration"]

df = pd.read_csv(data_file, sep='\s+', skiprows=5, header=None,
skipfooter=1, engine='python')
df.columns = data_columns

df = df.assign(**df.apply(convert_to_kilobytes, axis=1))

plt.figure(figsize=(10, 6))
sns.histplot(df['total_frames'], kde=False)
plt.title('Flow Length Distribution (Total Frames)')
plt.xlabel('Flow Length (Frames)')
plt.ylabel('Frequency')
plt.yscale('log')
plt.grid(axis='y')
plt.show()

plt.figure(figsize=(10, 6))
sns.ecdfplot(df['total_frames'])
```

```
plt.title('ECDF of Flow Length')
plt.xlabel('Flow Length (Frames)')
plt.ylabel('ECDF')
plt.ylim(0, 1.2)
plt.grid(True)
plt.show()

print("Key Summary Statistics (Total Frames):")
print(df['total_frames'].describe())
```

This code primarily focuses on reading and processing network traffic data, converting the byte units in the data to kilobytes, and performing a visual analysis. It begins by reading traffic data from a text file and then converts it into a unified kilobyte unit. Following that, the code utilizes histograms and Empirical Cumulative Distribution Function (ECDF) plots to display the distribution of the total number of traffic frames, and it prints out key statistical information about the total frame count.





The first graph presents the flow length distribution on a logarithmic scale, showing the frequency of network flows of varying lengths. The distribution is skewed towards shorter flows, indicating that most network flows are relatively short in length. As flow length increases, the frequency significantly decreases, which is typical for many networks where the bulk of flows are small. The second graph depicts the empirical cumulative distribution function (ECDF) for flow lengths. This graph shows that a large proportion of flows are short, as the ECDF rapidly rises to 1 at the lower end of the flow length spectrum.

```
Key Summary Statistics:
count      1019.000000
mean        384.937193
std        5627.409186
min           1.000000
25%          15.000000
50%          24.000000
75%          35.000000
max      145777.000000
```

9. Fit a distribution for the flow lengths and validate the model.

```
import pandas as pd
import matplotlib.pyplot as plt
from distfit import distfit
```

```
def convert_to_byte(row):
    units = {'bytes': 1, 'kb': 1024, 'mb': 1024**2}

    try:
        ld_bytes_unit = str(row['ld_bytes_unit']).lower()
        factor = units[ld_bytes_unit]
        ld_kb = row['ld_bytes'] * factor

        rd_bytes_unit = str(row['rd_bytes_unit']).lower()
        factor = units[rd_bytes_unit]
        rd_kb = row['rd_bytes'] * factor

        total_bytes_unit = str(row['total_bytes_unit']).lower()
        factor = units[total_bytes_unit]
        total_kb = row['total_bytes'] * factor

        return pd.Series({'ld_bytes': ld_kb, 'rd_bytes': rd_kb,
'total_bytes': total_kb, 'server_ip': row['second_ip_interface']})
    except KeyError as e:
        print(f"Error processing row {row}: {e}")
        raise ValueError("Invalid unit. Supported units are 'bytes',
'kb', 'mb.")

df = pd.read_csv('files/final.txt', sep='\s+', skiprows=5,
header=None, skipfooter=1, engine='python')

new_column_names = ["first_ip_interface", "arrow",
"second_ip_interface", "ld_frames", "ld_bytes", "ld_bytes_unit",
                    "rd_frames", "rd_bytes", "rd_bytes_unit",
"total_frames", "total_bytes", "total_bytes_unit",
                    "start", "duration"]

df.columns = new_column_names
pd.set_option('display.max_columns', None)
df = df.assign(**df.apply(convert_to_byte, axis=1))
data = df['total_frames']
dist = distfit()
dist.fit_transform(data)
```

```
print(dist.summary)
dist.plot()
plt.show()
```

This code is designed to process and analyze network traffic data, with a focus on converting data units and fitting a specific data column to a probability distribution. Initially, it reads network traffic data from a text file, skipping certain lines for formatting purposes, and sets specific column names. Then, it uses a function to convert the data in various columns into a unified byte unit. Next, it employs the distfit library to fit the 'total_frames' column data to a probability distribution, followed by plotting the best-fit distribution. Finally, a summary is printed, which includes statistical details about the fitted distribution.

```
[distfit] >INFO> fit
[distfit] >INFO> transform
[distfit] >INFO> [norm      ] [0.00 sec] [RSS: 0.00492547] [loc=384.937 scale=5624.647]
[distfit] >INFO> [expon     ] [0.00 sec] [RSS: 0.00411874] [loc=1.000 scale=383.937]
[distfit] >INFO> [pareto    ] [0.00 sec] [RSS: 0.00196315] [loc=-35.452 scale=36.452]
[distfit] >INFO> [dweibull  ] [0.03 sec] [RSS: 0.00289746] [loc=23.000 scale=557.047]
[distfit] >INFO> [t         ] [0.09 sec] [RSS: 0.000615548] [loc=22.093 scale=8.464]
[distfit] >INFO> [genextreme] [0.05 sec] [RSS: 0.00537514] [loc=3.183 scale=5.457]
[distfit] >INFO> [gamma     ] [0.03 sec] [RSS: 0.00447127] [loc=1.000 scale=75390.558]
[distfit] >INFO> [lognorm   ] [0.00 sec] [RSS: 0.00355599] [loc=1.000 scale=3.752]
[distfit] >INFO> [beta      ] [0.07 sec] [RSS: 0.00266642] [loc=1.000 scale=403680.218]
[distfit] >INFO> [uniform   ] [0.00 sec] [RSS: 0.00495236] [loc=1.000 scale=145776.000]
[distfit] >INFO> [loggamma  ] [0.02 sec] [RSS: 0.00493174] [loc=-3922841.817 scale=470315.668]
```

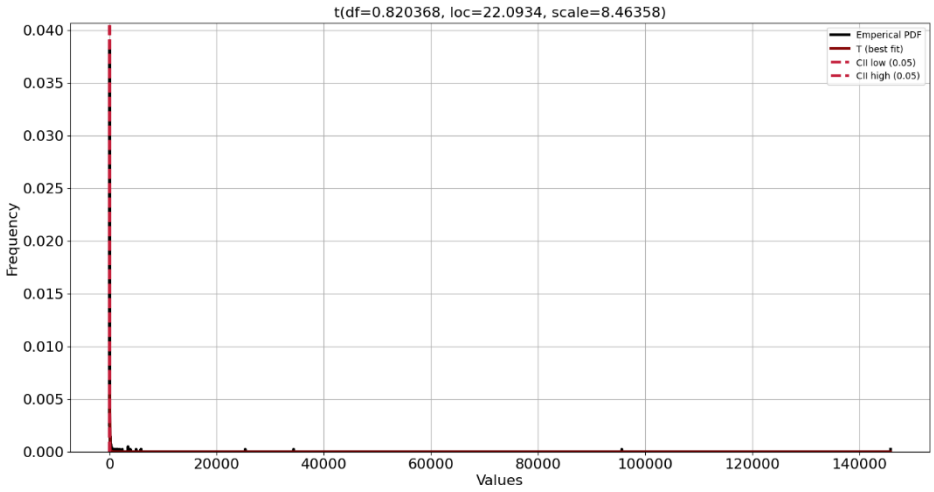
	name	score	loc	scale	\
0	t	0.000616	22.093362	8.463583	
1	pareto	0.001963	-35.452159	36.452159	
2	beta	0.002666	1.0	403680.218213	
3	dweibull	0.002897	23.0	557.046797	
4	lognorm	0.003556	1.0	3.75164	
5	expon	0.004119	1.0	383.937193	
6	gamma	0.004471	1.0	75390.557741	
7	norm	0.004925	384.937193	5624.647267	
8	loggamma	0.004932	-3922841.816552	470315.668362	
9	uniform	0.004952	1.0	145776.0	
10	genextreme	0.005375	3.182655	5.456834	

	arg
0	(0.8203678345199359,)
1	(1.4713195879370295,)
2	(0.337799544194041, 765.4276430934046)
3	(0.3542081928546171,)
4	(8.674642176068659,)
5	()
6	(0.013093758177759757,)
7	()
8	(4194.755188170506,)
9	()
10	(-2.3961918759224723,)

	params
0	(0.8203678345199359, 22.09336168433228, 8.4635...
1	(1.4713195879370295, -35.452159333990046, 36.4...
2	(0.337799544194041, 765.4276430934046, 0.99999...
3	(0.3542081928546171, 22.999999999999993, 557.0...
4	(8.674642176068659, 0.9999999999999998, 3.7516...
5	(1.0, 383.937193326791)
6	(0.013093758177759757, 0.9999999999999998, 753...
7	(384.937193326791, 5624.647267098104)
8	(4194.755188170506, -3922841.8165521715, 47031...
9	(1.0, 145776.0)
10	(-2.3961918759224723, 3.1826548679954687, 5.45...

	model	bootstrap_score
0	<scipy.stats._distn_infrastructure.rv_continuo...	0
1	<scipy.stats._distn_infrastructure.rv_continuo...	0
2	<scipy.stats._distn_infrastructure.rv_continuo...	0
3	<scipy.stats._distn_infrastructure.rv_continuo...	0
4	<scipy.stats._distn_infrastructure.rv_continuo...	0
5	<scipy.stats._distn_infrastructure.rv_continuo...	0
6	<scipy.stats._distn_infrastructure.rv_continuo...	0
7	<scipy.stats._distn_infrastructure.rv_continuo...	0
8	<scipy.stats._distn_infrastructure.rv_continuo...	0
9	<scipy.stats._distn_infrastructure.rv_continuo...	0
10	<scipy.stats._distn_infrastructure.rv_continuo...	0

	bootstrap_pass	color
0	None	#e41a1c
1	None	#e41a1c
2	None	#377eb8
3	None	#4daf4a
4	None	#984ea3
5	None	#ff7f00
6	None	#ffff33
7	None	#a65628
8	None	#f781bf
9	None	#999999
10	None	#999999



10. Compare the number of flows with 1, 10, 60, 120 and 1800 second timeouts. In this, you need to generate flow data multiple times.

```
from scapy.all import rdpcap, PacketList
from datetime import datetime

# Function to analyze TCP flows based on different timeout values
def analyze_tcp_flows(pcap_file, timeouts):
    flows_count = {}
    for timeout in timeouts:
        flows = []
        current_flow_packets = []
        last_packet_time = None
```

```
        for packet in rdpcap(pcap_file):
            if 'IP' in packet and 'TCP' in packet:
                packet_time =
datetime.fromtimestamp(float(packet.time))

                if last_packet_time is None or (packet_time -
last_packet_time).seconds > timeout:
                    if current_flow_packets:
                        flows.append(PacketList(current_flow_packets))
                    current_flow_packets = [packet]
                else:
                    current_flow_packets.append(packet)

                last_packet_time = packet_time

        if current_flow_packets:
            flows.append(PacketList(current_flow_packets))

        flows_count[timeout] = len(flows)
    return flows_count

# Main code
pcap_file_path = 'files/final.pcap'
timeout_seconds = [1, 10, 60, 120, 1800]
flows_count_by_timeout = analyze_tcp_flows(pcap_file_path,
timeout_seconds)

for timeout, flow_count in flows_count_by_timeout.items():
    print(f"Timeout: {timeout} seconds, Flow Count: {flow_count}")
```

This code starts by reading a PCAP file and then iterates through each packet. It uses a series of predefined timeout thresholds (in seconds) to determine whether packets belong to the same flow. If the time difference between two consecutive packets exceeds the current timeout threshold, they are considered to belong to different flows. In this way, the code can count the number of flows for each timeout threshold and print the results.

Aalto University
School of Electrical
Engineering

```
Timeout: 1 seconds, Flow Count: 2341
Timeout: 10 seconds, Flow Count: 1
Timeout: 60 seconds, Flow Count: 1
Timeout: 120 seconds, Flow Count: 1
Timeout: 1800 seconds, Flow Count: 1
```

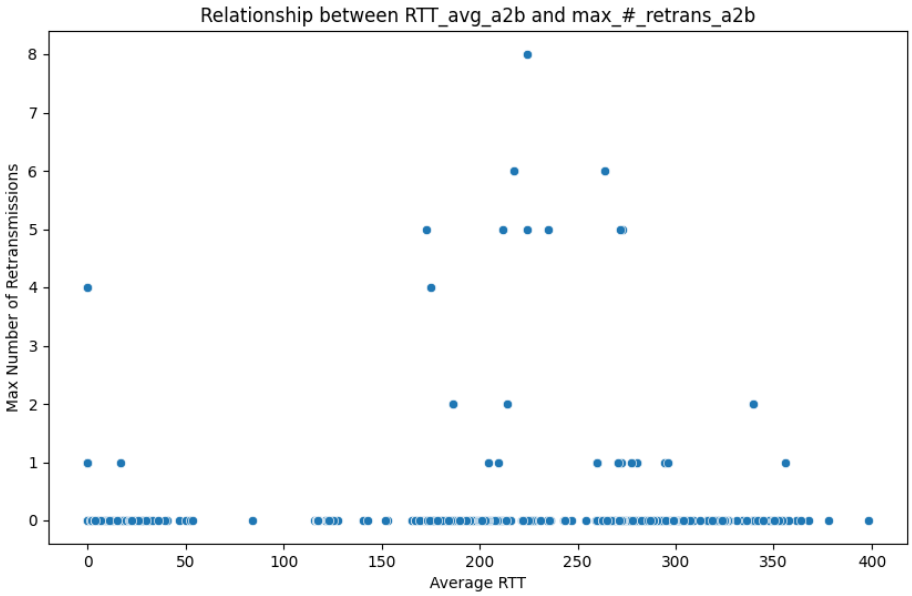
- TCP connection data PS3

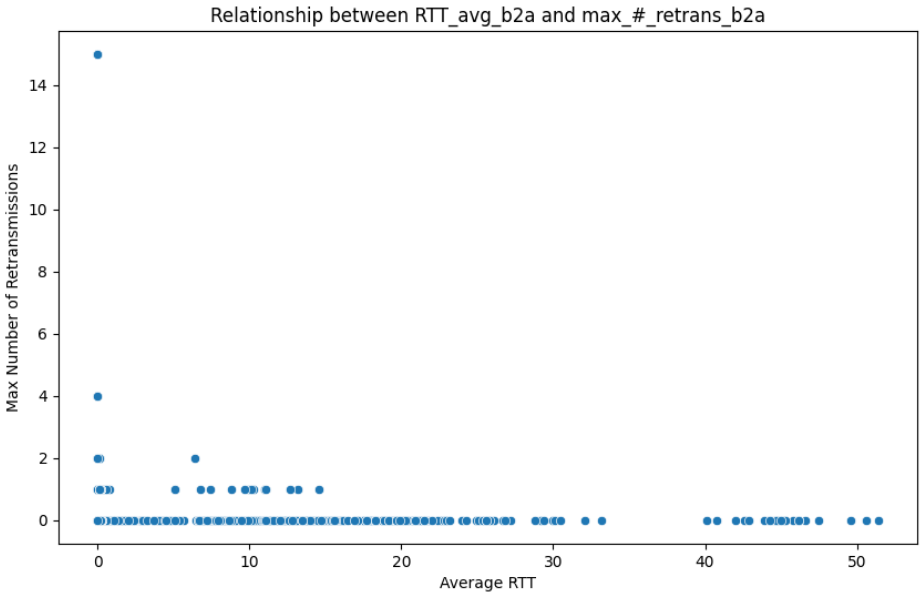
11. Round-trip times and their variance.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

def load_and_clean_data(file_path, columns_to_clean):
    data = pd.read_csv(file_path)
    data_cleaned = data.dropna(subset=columns_to_clean)
    return data_cleaned

def plot_relationship(data, x_column, y_column):
    plt.figure(figsize=(10, 6))
    sns.scatterplot(x=data[x_column], y=data[y_column])
    plt.title(f'Relationship between {x_column} and {y_column}')
    plt.xlabel('Average RTT (ms)')
    plt.ylabel('Max Number of Retransmissions')
    plt.show()

    # Calculate and print correlation, mean, and variance
    correlation = data[x_column].corr(data[y_column])
    print(f"Correlation between {x_column} and {y_column}:
{correlation:.2f}")
    mean_value = data[y_column].mean()
    variance_value = data[y_column].var()
    print(f"Mean of {y_column}: {mean_value:.2f}")
    print(f"Variance of {y_column}: {variance_value:.2f}")

# Main code
csv_file_path = 'files/finaltcp.csv'
```

```
columns_to_clean = ['RTT_avg_a2b', 'RTT_avg_b2a',
'max_#_retrans_a2b', 'max_#_retrans_b2a']
tcp_traffic_data = load_and_clean_data(csv_file_path,
columns_to_clean)

# Column names for average RTT and max number of retransmissions
rtt_avg_columns = ['RTT_avg_a2b', 'RTT_avg_b2a']
retrans_max_columns = ['max_#_retrans_a2b', 'max_#_retrans_b2a']

# Analyze the relationship between RTT and retransmissions
for rtt_col, retrans_col in zip(rtt_avg_columns,
retrans_max_columns):
    plot_relationship(tcp_traffic_data, rtt_col, retrans_col)
```

The primary function of this code is to analyze and visualize the relationship between the Average Round-Trip Time (RTT) and the Maximum Number of Retransmissions in network traffic data. It starts by reading data from a CSV file, then conducts analysis on two pairs of columns. For each pair, the code generates a scatter plot to display the relationship between the average RTT and the maximum number of retransmissions and calculates the correlation coefficient between them.






```
Correlation between RTT_avg_a2b and max_#_retrans_a2b: -0.009378461122759854
Mean of max_#_retrans_a2b: 0.10607521697203472
Variance of max_#_retrans_a2b: 0.3632545619045138
Correlation between RTT_avg_b2a and max_#_retrans_b2a: -0.06492110182176081
Mean of max_#_retrans_b2a: 0.14754098360655737
Variance of max_#_retrans_b2a: 1.1934616114943988
```

The first two graphs display the relationship between the Average Round-Trip Time (RTT) and the Maximum Number of Retransmissions for two directions, from address A to B (a2b) and from address B to A (b2a), respectively. In both scatter plots, there is no strong and consistent relationship evident. Most data points are clustered at the lower end of the RTT scale, indicating that the majority of communications have a low RTT and few retransmissions.

The statistical output provided in the third graph offers the correlation coefficients between RTT and the maximum number of retransmissions for both directions. The correlations are very weak, suggesting no significant linear relationship between RTT and the maximum number of retransmissions. The average values and variances for the maximum number of retransmissions are quite low, indicating that high retransmissions are not common.

12. Total traffic volume during the connection (you get the volume from PS2).

```
import pandas as pd

def convert_units(row):
    # Define conversion factors for each unit
    unit_factors = {'bytes': 1, 'kb': 1024, 'mb': 1024 ** 2}
    converted_data = {}

    # Process each data type and convert to bytes
    for data_type in ['ld', 'rd', 'total']:
        bytes_value = row[f'{data_type}_bytes']
        unit = row[f'{data_type}_bytes_unit'].lower()

        # Raise an error if the unit is not recognized
        if unit not in unit_factors:
            raise ValueError(f"Invalid unit '{unit}' for
{data_type}_bytes. Supported units: 'bytes', 'kb', 'mb'.")

        # Perform conversion and store in the dictionary
        converted_data[f'{data_type}_bytes_converted'] = bytes_value *
unit_factors[unit]

    # Return a new series including the converted data and server IP
    return pd.Series({**converted_data, 'server_ip':
row['second_ip']})

# Read and preprocess data from a text file
data_file = 'files/final.txt'
column_names = [
    "first_ip", "arrow", "second_ip", "ld_frames", "ld_bytes",
"ld_bytes_unit",
    "rd_frames", "rd_bytes", "rd_bytes_unit", "total_frames",
"total_bytes", "total_bytes_unit",
    "start_time", "duration"
]

# Load the data, skipping the first 5 rows and the last row
df = pd.read_csv(
    data_file, sep='\s+', skiprows=5, header=None, skipfooter=1,
engine='python', names=column_names
)
```

```
# Apply the conversion function to each row
converted_df = df.apply(convert_units, axis=1)

# Calculate the total traffic volume in bytes
total_traffic_volume_bytes =
converted_df['total_bytes_converted'].sum()
print(f"The total traffic volume during the connection is:
{total_traffic_volume_bytes} bytes")
```

This code primarily focuses on processing and analyzing network traffic data. It reads traffic data from a text file and then uses a function to convert the byte units in the data into a unified byte format. After the conversion, it calculates the total bytes across all rows in the dataset and prints out this sum.

```
The total traffic volume during the connection is: 385867549
```

- Conclusions

  - Traffic volume at different time scales. Are there any identifiable patterns or trends that you observed?

    The packet traffic volume graphs, calculated by the second and by the minute, show significant spikes in traffic at specific times, indicating that a large amount of data transfer or communication activity may occur at these moments. These peaks are more pronounced and easier to identify on a per-second basis, as traffic can vary greatly within a single second.

    Similarly, the flow data graphs also display peaks in traffic volume. However, the peaks in flow data occur less frequently but with greater volume. This suggests that when flows occur, they may involve the transfer of larger amounts of data at once, or multiple packets may be combined into flows.

    Overall, the traffic volume measured by the second reveals more detail and variability, as packet transmission can be bursty. On a per-minute scale, the overall trend becomes

smoother, potentially averaging out the extreme values seen on a per-second basis, which could be more useful for identifying long-term patterns or trends in network traffic.

- The top 5 most common applications based on their port numbers. Identify the corresponding applications (e.g., HTTPS application) and analyze their characteristics.

  Top 5 port numbers: 443, 61784, 52330, 61788, 5259.

  Port 443: This is the standard port for HTTPS traffic, which is the secure version of HTTP. The characteristic of this application is that it encrypts communication to ensure secure transactions, typically used for online banking, e-commerce, and any other service where data security is paramount.

  Port 61784: Likely used by a custom application or service. The application could be proprietary or specific to a certain vendor, possibly for remote service, gaming, or peer-to-peer applications.

  Port 52330: Similar to port 61784, this is likely used by a private service or application. It might be used for internal communication within a private network, for instance, by backend systems of a web service.

  Port 61788: As with the other dynamic ports, this port is probably assigned dynamically for a specific session or service. Applications using such ports often include streaming services, virtual private networks (VPN), or other temporary communication sessions.

  Port 5259: This port might be employed by a specific application, such as a multimedia streaming service or a networked game that requires real-time data transmission.

- Differences of flow and packet measurements in the example case.

  Packet measurement provides a magnified view of network traffic, while flow measurement offers a broader view of extended communication patterns. The fine-grained nature of packet analysis is particularly suitable for detailed troubleshooting and performance analysis. In contrast, flow analysis can help identify trends over longer periods, understand the nature of the traffic, and plan network capacity. High peaks in

packet data may reflect transient events or anomalies, whereas the smoother trends in flow data might reflect regular usage patterns or scheduled activities.

- Your findings on retransmissions.

The low average values of retransmissions indicate that most communications in network traffic experience few retransmissions. The weak negative correlation between RTT and retransmissions may suggest that longer RTTs are not necessarily associated with an increased number of retransmissions. This could imply that other factors, such as network congestion, packet loss, or connection quality, might have a more significant impact on retransmissions. The scatter plots show no clear trend, indicating that retransmissions are not solely influenced by RTT and are likely affected by a complex set of factors.

Aalto University
School of Electrical
Engineering

Task 2: Flow data

- Acquiring flow data
- Data pre-processing
  - Subnetwork: 163.35.138.0/24
  - Commands or code that is used in pre-processing.

    gawk '$1~/^163\.35\.138\./||$15~/^163\.35\.138\./' 1200.t2 > ~/my_1200.t2
    The contents of the commands are the same, only the file names are different.

  - Short samples (10 lines or so) taken from the distilled data.

```
48      6       1       55402   443     12      1767    1       1491988858.280602000    1491988859.099810000
163.35.138.122  204.153.144.148 6       1       55401   443     17      2187    1       1491988858.275957000    1491988859.478691000
163.35.138.122  204.153.144.148 6       1       55400   443     11      1747    1       1491988843.903724000    1491988844.679525000
163.35.138.122  204.153.144.148 6       1       55399   443     10      1732    1       1491988843.213457000    1491988843.899046000
163.35.138.122  204.153.144.148 6       1       55398   443     12      1723    1       1491988843.183613000    1491988843.937466000
163.35.138.122  204.153.144.148 6       1       55397   443     10      1688    1       1491988842.254293000    1491988842.961055000
163.35.138.122  204.153.144.148 6       1       55396   443     11      1753    1       1491988841.486457000    1491988842.223009000
163.35.138.122  204.153.144.148 6       1       55395   443     15      1897    1       1491988838.047840000    1491988841.419759000
163.35.138.122  204.153.144.148 6       1       55394   443     11      1754    1       1491988836.056238000    1491988838.043909000
163.35.138.122  204.153.144.148 6       1       55393   443     12      1777    1       1491988832.758846000    1491988834.025902000
163.35.138.122  204.153.144.148 6       1       55392   443     11      1757    1       1491988831.796465000    1491988832.541608000
```

- Data analysis

1. Plot traffic volume

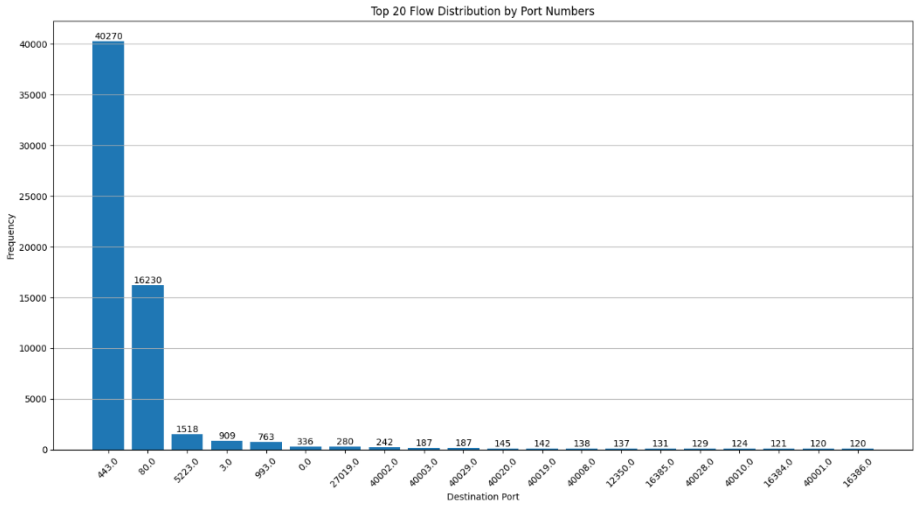
   - Visualise flow distribution by port numbers.

```
import pandas as pd
import matplotlib.pyplot as plt
import glob

def read_and_combine_data(directory, file_pattern, column_names):
    data_frames = []
    for file_path in glob.glob(directory + file_pattern):
        try:
            data_frame = pd.read_csv(file_path, sep='\t', header=None,
```

```
names=column_names)
            data_frames.append(data_frame)
        except pd.errors.ParserError as error:
            print(f"Error reading {file_path}: {error}")
    return pd.concat(data_frames, ignore_index=True) if data_frames
else pd.DataFrame()

def plot_top_destination_ports(data, top_n=20):
    destination_port_counts =
data['destination_port'].value_counts().sort_values(ascending=False).
head(top_n)
    plt.figure(figsize=(10, 8))
    bar_plot = plt.bar(destination_port_counts.index.astype(str),
destination_port_counts.values)

    for bar in bar_plot:
        height = bar.get_height()
        plt.text(bar.get_x() + bar.get_width() / 2, height,
int(height), ha='center', va='bottom')

    plt.grid(axis='y')
    plt.title('Top 20 Flow Distribution by Port Numbers')
    plt.xlabel('Destination Port')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45)
    plt.show()

# Path to the directory containing the data files
data_dir = 'files/data/'

# Column names for the data
column_names = ['source', 'destination', 'protocol', 'status_ok',
'source_port',
                'destination_port', 'packet_count', 'byte_count',
'flow_count',
                'first_seen', 'last_seen']

# Combine data from all files
combined_data = read_and_combine_data(data_dir, '*.t2', column_names)
```

```
# Check if DataFrame is not empty before plotting
if not combined_data.empty:
    plot_top_destination_ports(combined_data)
else:
    print("No data available for plotting.")
```

The purpose of this code is to read and consolidate data from multiple files within a specified directory, where each file contains tab-separated values with specific column names. It then analyzes and visualizes the top destination ports by frequency, displaying the distribution of flow data for these ports in a bar chart.



This chart shows the frequency distribution of network or computer system traffic across different destination ports. The frequencies for ports "80" and "443" are significantly higher than those for other ports. This is consistent with standard network traffic, as port 80 is typically used for HTTP traffic and port 443 for HTTPS traffic. The frequencies for the other ports are much lower, indicating less traffic.

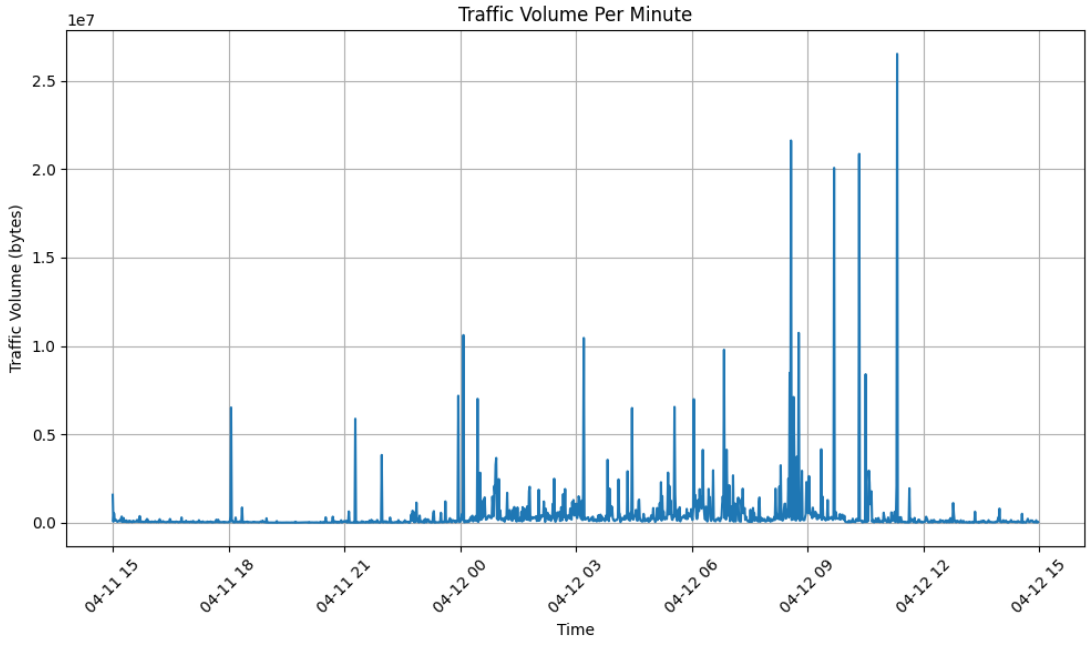
- Plot traffic volume as a function of time with at least two sufficiently different time scales.

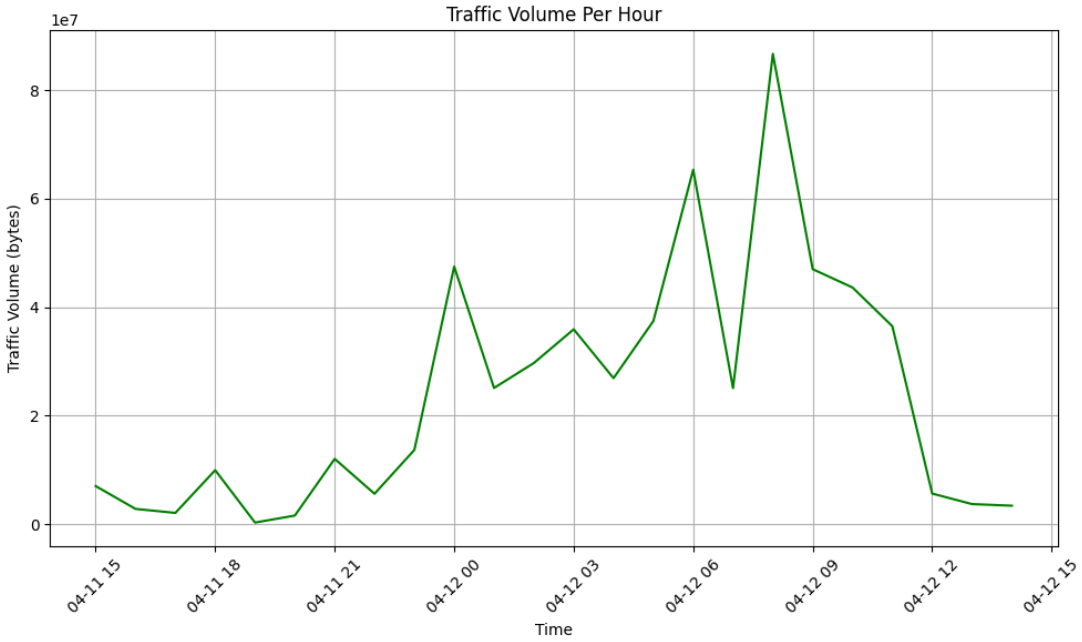
```
import pandas as pd
import matplotlib.pyplot as plt
import glob
```

```
def read_and_process_data(directory, file_pattern, column_headers):
    file_paths = glob.glob(directory + file_pattern)
    dataframes = []
    for file_path in file_paths:
        try:
            data = pd.read_csv(file_path, sep='\t', header=None,
names=column_headers)
            if not data.empty:
                dataframes.append(data)
        except pd.errors.ParserError as e:
            print(f"Skipping file {file_path} due to parsing error:
{e}")
    if not dataframes:
        raise ValueError("No data files found or all files are
empty.")
    combined_dataframe = pd.concat(dataframes, ignore_index=True)
    return combined_dataframe

def plot_traffic_volume(traffic_data, title, time_interval,
color='blue'):
    plt.figure(figsize=(10, 6))
    plt.plot(traffic_data, color=color)
    plt.grid(True)
    plt.xlabel('Time')
    plt.ylabel('Traffic Volume (bytes)')
    plt.title(f'Traffic Volume {title}')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

data_directory = 'files/data/'
data_file_pattern = '*.t2'
column_headers = ['source', 'destination', 'protocol', 'status_ok',
'source_port', 'dest_port', 'packet_count', 'byte_count',
'flow_count', 'start_time', 'end_time']

traffic_data_combined = read_and_process_data(data_directory,
data_file_pattern, column_headers)

traffic_data_combined['start_time'] =
```

```
pd.to_datetime(traffic_data_combined['start_time'], unit='s')

# Set the 'start_time' column as the index for resampling
traffic_data_combined.set_index('start_time', inplace=True)

# Resample to get traffic volume per minute and per hour
traffic_volume_per_minute =
traffic_data_combined.resample('T')['byte_count'].sum()
traffic_volume_per_hour =
traffic_data_combined.resample('H')['byte_count'].sum()

# Plot the traffic volume per minute and per hour
plot_traffic_volume(traffic_volume_per_minute, 'Per Minute',
'Minute')
plot_traffic_volume(traffic_volume_per_hour, 'Per Hour', 'Hour',
color='green')
```

The purpose of this code is to read and process data from multiple files located in a specified directory. These files contain tab-separated data with specific column headers. The code then conducts time series analysis by resampling the data into minute and hour intervals, calculating the total traffic volume. It subsequently plots two graphs: one displaying traffic volume changes per minute and another showing traffic volume changes per hour.





These two charts reflect network traffic variations across different time scales. The first chart, with per-minute granularity, shows the fluctuations in traffic from April 11 to April 15, with many sharp peaks indicating brief periods of high traffic activity. In contrast, the second chart displays the change in traffic on an hourly basis, revealing peaks in traffic volume within certain hours, followed by a gradual decline. Integrating the information from both charts, we can observe that network traffic experiences intense fluctuations at specific moments, while a more macroscopic time scale reveals the overall trends in network usage.

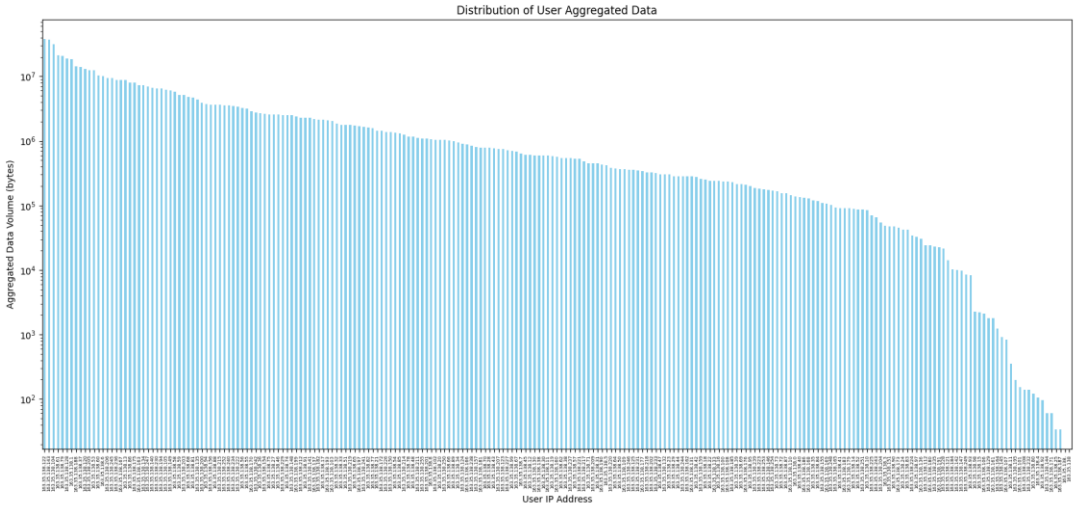
2. Per user data volume

```
import pandas as pd
import matplotlib.pyplot as plt
import glob
import numpy as np

def load_and_combine_data(directory):
    # Define column names
    columns = ['source_ip', 'destination_ip', 'protocol', 'status',
'source_port', 'destination_port',
               'packets', 'bytes', 'flows', 'first_timestamp',
'latest_timestamp']
```

```
    # Use glob to get all file paths in the directory
    file_paths = glob.glob(directory + '*.t2')

    # Read all files and combine into one DataFrame
    df_list = [pd.read_csv(file, sep='\t', header=None,
names=columns) for file in file_paths]
    return pd.concat(df_list, ignore_index=True)

def plot_user_data_volume(data, font_size=12):
    # Calculate the aggregated data volume for each user (source IP
address)
    user_data_volume =
data.groupby('source_ip')['bytes'].sum().sort_values(ascending=False)

    # Plot bar chart for the aggregated data volume of all users
    plt.figure(figsize=(12, 6))
    user_data_volume.plot(kind='bar', color='skyblue')
    plt.title('Distribution of User Aggregated Data')
    plt.xlabel('User IP Address')
    plt.ylabel('Aggregated Data Volume (bytes)')
    plt.xticks(rotation=90, fontsize=font_size)  # Rotate the x labels
and set font size
    plt.yscale('log')  # Use logarithmic scale for better visibility
    plt.tight_layout()  # Adjust layout to fit IP addresses
    plt.show()

# Define the directory path for data files
directory_path = 'files/data/'

# Load and combine data from files
df_combined = load_and_combine_data(directory_path)

# Plot the user data volume distribution
plot_user_data_volume(df_combined, font_size=5)
```

The functionality of this code is to load multiple data files with the `.t2` extension from a specified directory, merge them into a single DataFrame, calculate the total data volume for each user (source IP address), and plot a bar chart on a logarithmic scale to visualize the distribution of aggregated

data volume among users, facilitating data analysis and visualization.



This chart presents the data usage across different users. It reveals significant disparities in data consumption; some users have much higher data usage than others, while the majority of users consume relatively small amounts of data. The chart illustrates a long-tail distribution, where a small number of users account for the majority of the data volume, and the vast majority of users contribute a small fraction of the total usage.

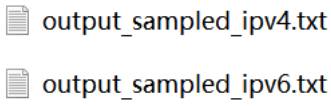
3. Flow sampling

- Shell for sampling:

```
#!/bin/bash

# Define the input directory and the sample size per file
input_directory="/work/courses/unix/T/ELEC/E7130/general/trace/flow-
continue"
sample_per_file=7126

# Define the output files for IPv4 and IPv6
output_ipv4_file="./output_sampled_ipv4.txt"
output_ipv6_file="./output_sampled_ipv6.txt"

# Regular expressions for IPv4 and IPv6
```

```
ipv4_regex="^([0-9]{1,3}\.){3}[0-9]{1,3}$"
ipv6_regex="^(([0-9a-fA-F]{1,4}:){7,7}[0-9a-fA-F]{1,4}|([0-9a-fA-
F]{1,4}:){1,7}:|([0-9a-fA-F]{1,4}:){1,6}:[0-9a-fA-F]{1,4}|([0-9a-fA-
F]{1,4}:){1,5}(:[0-9a-fA-F]{1,4}){1,2}|([0-9a-fA-F]{1,4}:){1,4}(:[0-
9a-fA-F]{1,4}){1,3}|([0-9a-fA-F]{1,4}:){1,3}(:[0-9a-fA-
F]{1,4}){1,4}|([0-9a-fA-F]{1,4}:){1,2}(:[0-9a-fA-F]{1,4}){1,5}|[0-9a-
fA-F]{1,4}:((:[0-9a-fA-F]{1,4}){1,6})|:((:[0-9a-fA-F]{1,4}){1,7}|:))"

# Remove existing output files
rm -f "$output_ipv4_file" "$output_ipv6_file"

# Get the total number of files
total_files=$(find "$input_directory" -type f | wc -l)
current_file=0

# Process each file
for file in "$input_directory"/*; do
    if [ -f "$file" ]; then
        let current_file++
        echo "Processing file ($current_file / $total_files): $file"

        # Create temporary files for IPv4 and IPv6
        temp_ipv4="./temp_ipv4"
        temp_ipv6="./temp_ipv6"

        # Clear or initialize these files
        > "$temp_ipv4"
        > "$temp_ipv6"

        # Split the file into IPv4 and IPv6 parts
        tail -n +29 "$file" | awk -v ipv4_regex="$ipv4_regex" '$1 ~
ipv4_regex' > "$temp_ipv4"
        tail -n +29 "$file" | awk -v ipv6_regex="$ipv6_regex" '$1 ~
ipv6_regex' > "$temp_ipv6"

        # Sample the IPv4 temporary file
        total_lines_ipv4=$(wc -l < "$temp_ipv4")
        if [ $total_lines_ipv4 -ge $sample_per_file ]; then
            selected_lines_ipv4=($(shuf -i 1-$total_lines_ipv4 -n
$sample_per_file))
```

```
            python3 sample_script.py "$temp_ipv4"
"${selected_lines_ipv4[@]}" >> "$output_ipv4_file"
        else
            cat "$temp_ipv4" >> "$output_ipv4_file"
        fi

        # Sample the IPv6 temporary file
        total_lines_ipv6=$(wc -l < "$temp_ipv6")
        if [ $total_lines_ipv6 -ge $sample_per_file ]; then
            selected_lines_ipv6=($(shuf -i 1-$total_lines_ipv6 -n
$sample_per_file))
            python3 sample_script.py "$temp_ipv6"
"${selected_lines_ipv6[@]}" >> "$output_ipv6_file"
        else
            cat "$temp_ipv6" >> "$output_ipv6_file"
        fi

        # No longer need to delete these files
        # rm -f "$temp_ipv4" "$temp_ipv6"
    fi
done

echo "Sampling completed, IPv4 results saved in $output_ipv4_file,
IPv6 results saved in $output_ipv6_file"
```



This Bash script processes a collection of data files containing network traffic information, separating IPv4 and IPv6 traffic based on regular expressions, sampling a specified number of lines from each, and saving the sampled data into separate output files.

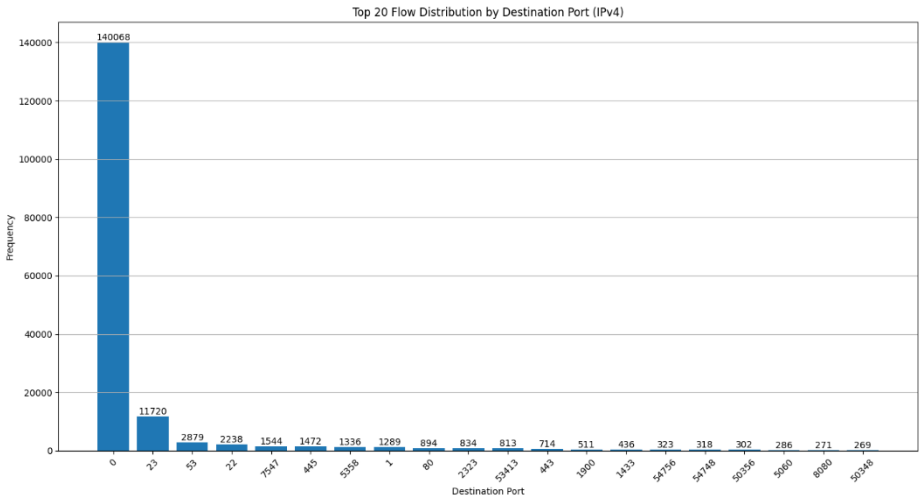
- Visualise flow distribution by port numbers.

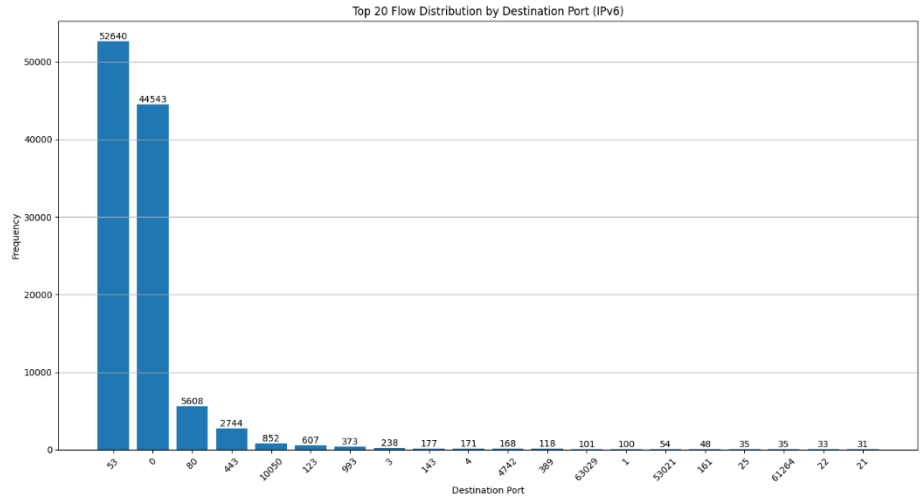
```
import pandas as pd
import matplotlib.pyplot as plt
import glob
```

```
# Define the file path for IPv6 traffic data
ipv6_traffic_file_path = 'files/output_sampled_ipv6.txt'

# Define descriptive column names
column_names = ['Source_IP', 'Destination_IP', 'Protocol',
'Status_OK', 'Source_Port', 'Destination_Port',
             'Packet_Count', 'Byte_Count', 'Flow_Count',
'Start_Time', 'End_Time']

# Read the IPv6 traffic data and create a DataFrame
ipv6_traffic_df = pd.read_csv(ipv6_traffic_file_path, sep='\t',
header=None, names=column_names)

# Group the data by 'Destination_Port', calculate counts, and sort in
descending order
top_destination_ports =
ipv6_traffic_df['Destination_Port'].value_counts().nlargest(20)

# Create a bar chart
plt.figure(figsize=(12, 8))
bars = plt.bar(top_destination_ports.index.astype(str),
top_destination_ports.values)

# Add text annotations above the bars
for bar in bars:
    y_value = bar.get_height()
    plt.text(bar.get_x() + bar.get_width() / 2, y_value,
int(y_value), ha='center', va='bottom')

plt.grid(axis='y')
plt.title('Top 20 Flow Distribution by Destination Port (IPv6)')
plt.xlabel('Destination Port')
plt.ylabel('Frequency')
plt.xticks(rotation=45)
plt.show()
```

The functionality of this code is to read IPv4/6 network traffic data files, parse the data within them, and extract destination port information. It then counts and plots the distribution of traffic for the top 20 destination ports, facilitating the analysis and visualization of usage patterns for IPv4/6 traffic

data.





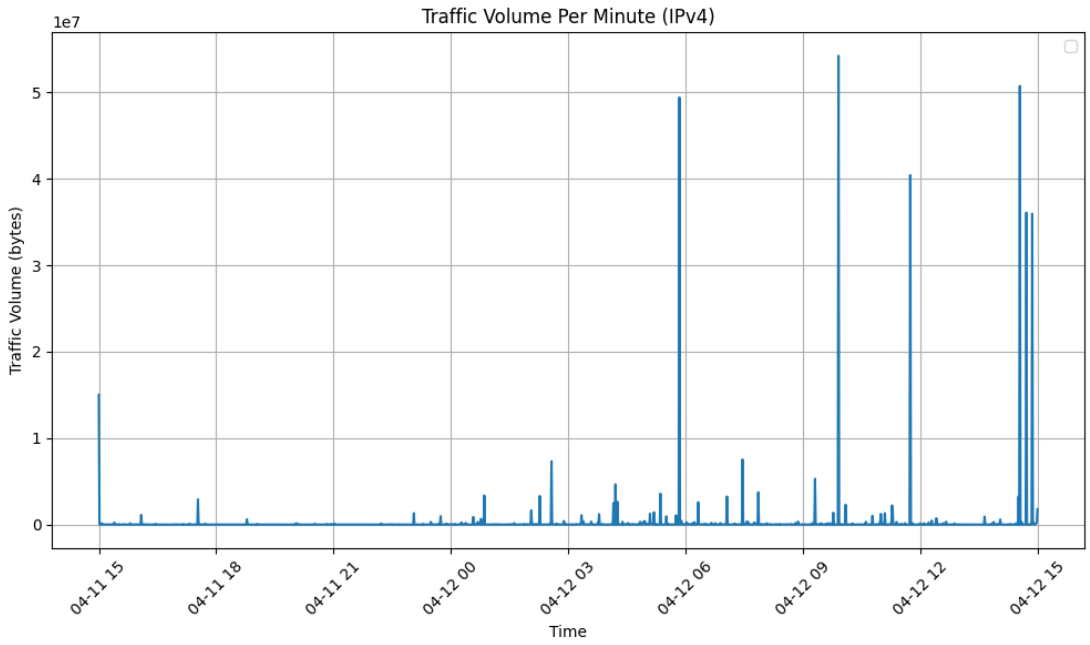
These two charts respectively display the top 20 flow distributions by destination port for IPv4 and Ipv6 traffic. In the Ipv4 chart, the flow peak for the first port exceeds 140,000, significantly higher than the other ports. The second port has about 11,720 flows, followed by a more gradual decline in traffic for the subsequent ports. This indicates that Ipv4 traffic is heavily concentrated on a specific port. In the Ipv6 chart, the flow distribution among the top ports is relatively more uniform.

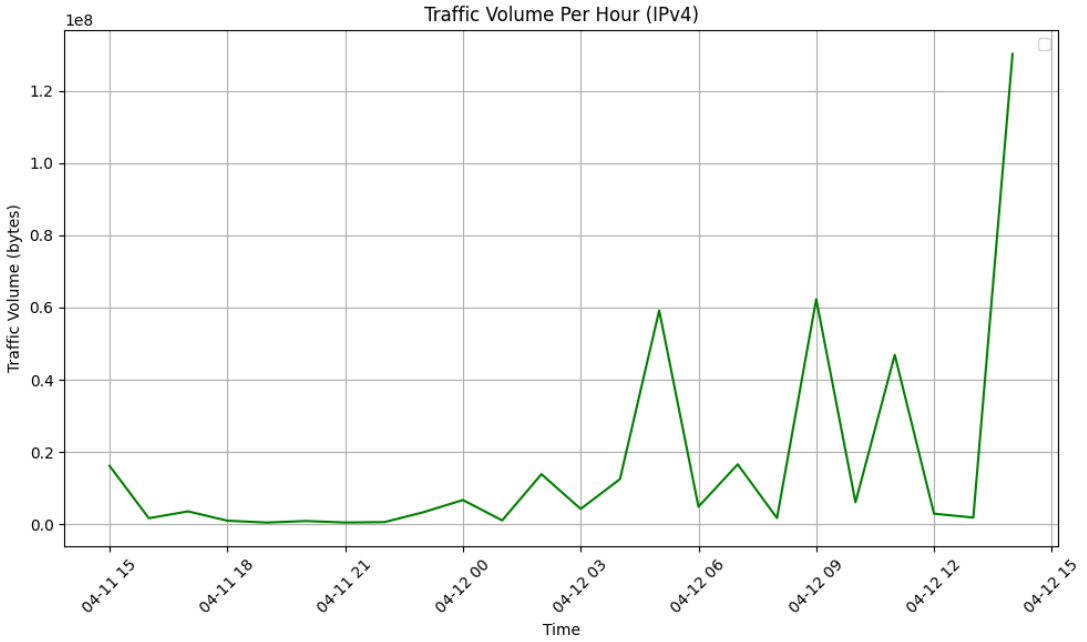
- Plot traffic volume as a function of time with at least two sufficiently different time scales.

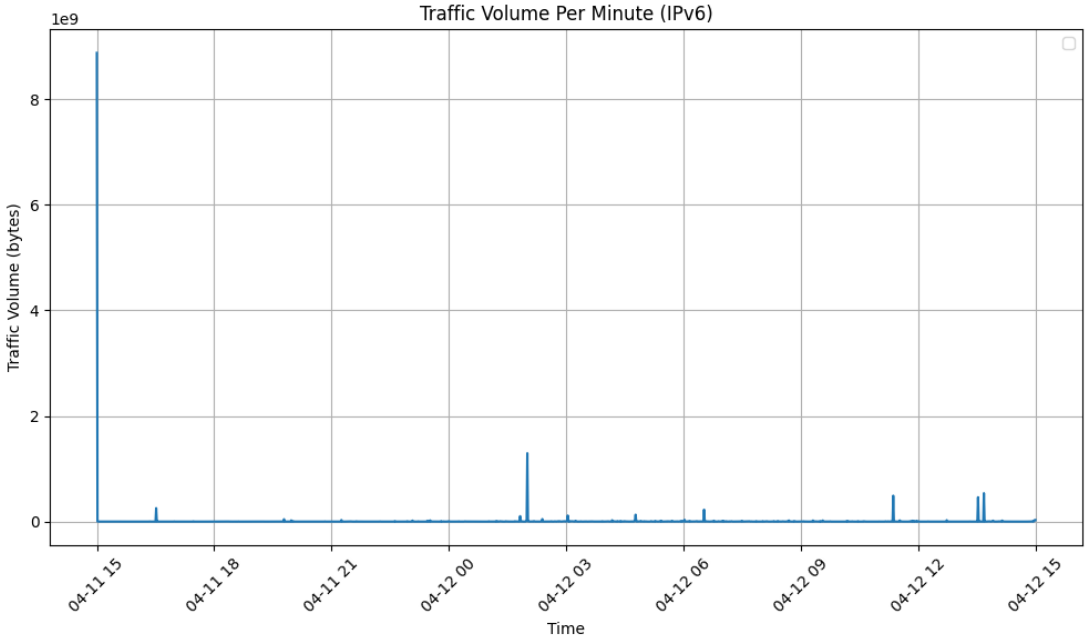
```
Import pandas as pd
import matplotlib.pyplot as plt

data_directory = 'files/output_sampled_ipv6.txt'
column_names = ['Source', 'Destination', 'Protocol', 'Status_OK',
'Source_Port', 'Destination_Port',
             'Packet_Count', 'Byte_Count', 'Flow_Count',
'Start_Time', 'End_Time']

traffic_data = pd.read_csv(data_directory, sep='\t', header=None,
names=column_names)

traffic_data['Start_Time'] =
pd.to_datetime(traffic_data['Start_Time'], unit='s')

traffic_data.set_index('Start_Time', inplace=True)
traffic_volume_per_minute =
traffic_data.resample('T')['Byte_Count'].sum()
traffic_volume_per_hour =
traffic_data.resample('H')['Byte_Count'].sum()

plt.figure(figsize=(10, 6))
plt.plot(traffic_volume_per_minute)
plt.grid(True)
plt.xlabel('Time')
plt.ylabel('Traffic Volume (bytes)')
plt.title('Traffic Volume Per Minute (Ipv6)')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(traffic_volume_per_hour, color='green')
plt.grid(True)
plt.xlabel('Time')
plt.ylabel('Traffic Volume (bytes)')
plt.title('Traffic Volume Per Hour (Ipv6)')
plt.xticks(rotation=45)
```

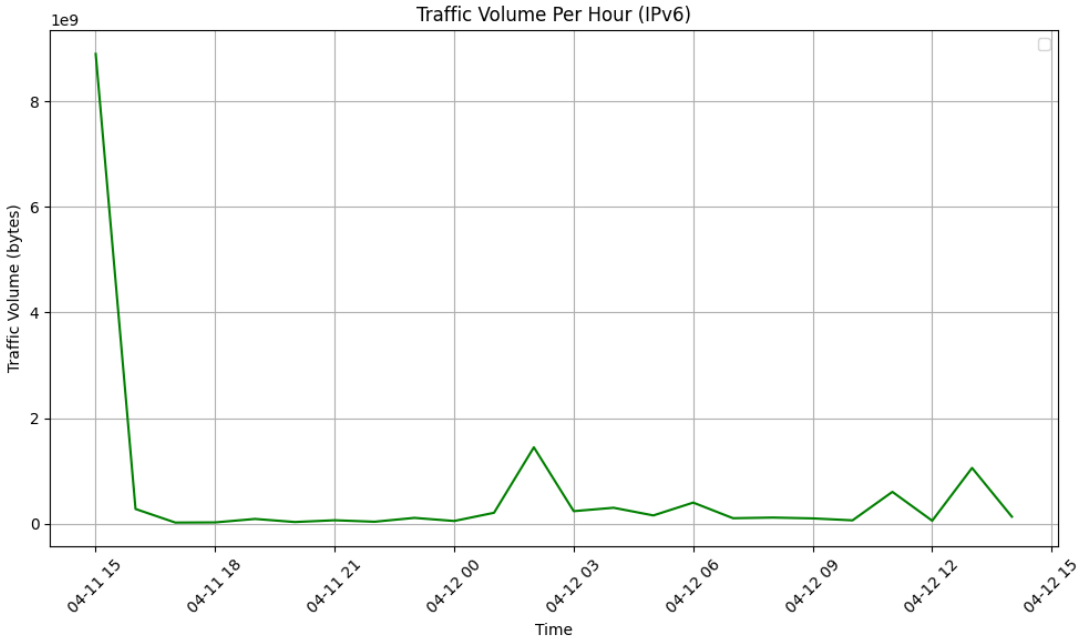
```
plt.tight_layout()
plt.show()
```

The functionality of this code is to read and preprocess IPv6 network traffic data files and then create two plots, each depicting the traffic trends over time. One plot represents the traffic trends summarized on a per-minute basis, while the other represents traffic trends summarized on an hourly basis.




Traffic Volume Per Hour (IPv4)
1e8
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Traffic Volume (bytes)
04-11 15
04-11 18
04-11 21
04-12 00
04-12 03
04-12 06
04-12 09
04-12 12
04-12 15
Time


Traffic Volume Per Minute (IPv6)
1e9
8
6
4
2
0
Traffic Volume (bytes)
04-11 15
04-11 18
04-11 21
04-12 00
04-12 03
04-12 06
04-12 09
04-12 12
04-12 15
Time



The IPv4 per-minute traffic volume chart shows several peaks, indicating surges in traffic during specific minutes. At other times, the traffic volume remains at a lower level. The per-hour traffic chart shows a smoother trend in volume changes. Although fluctuations are still observable, the difference between peaks and troughs is not as pronounced as in the per-minute chart.

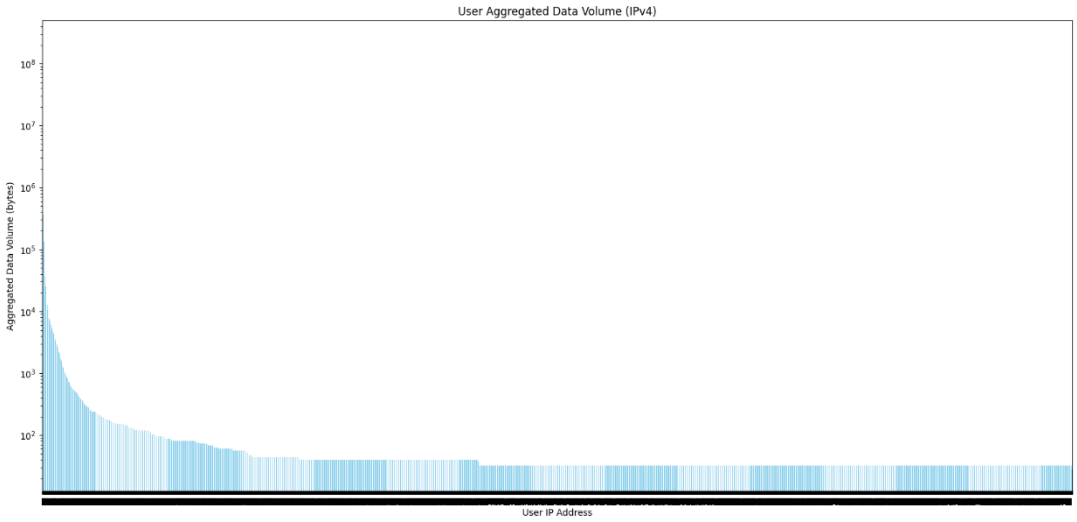
The IPv6 per-minute traffic volume chart exhibits characteristics similar to IPv4, with spikes in traffic volume occurring during certain minutes, though these spikes appear less frequently than in IPv4. The per-hour traffic chart displays a significant initial peak, after which the traffic volume quickly decreases and maintains a lower level, with occasional minor fluctuations. Compared to IPv4, the IPv6 traffic distribution seems more uniform, lacking many of the dramatic fluctuations.

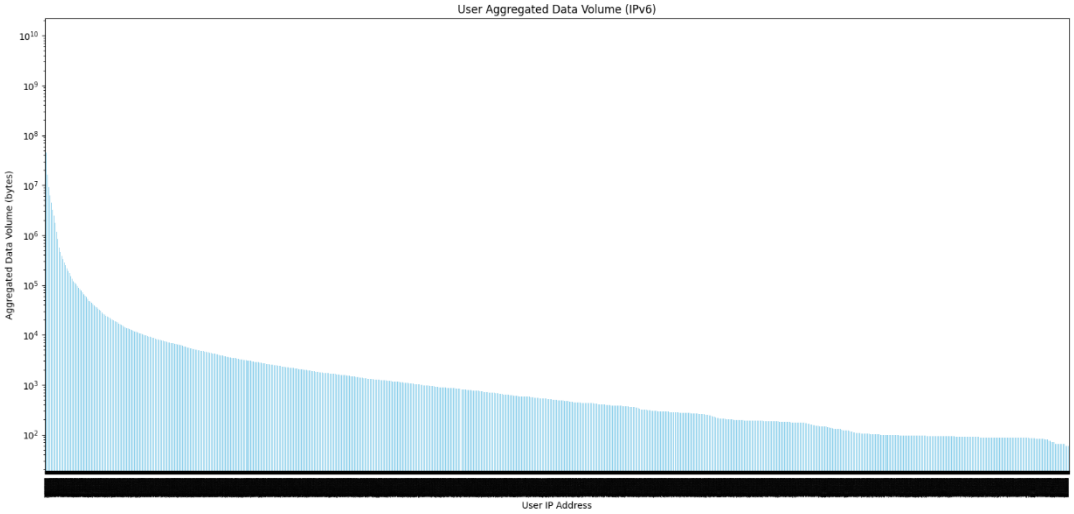
- Per user data volume

```
import pandas as pd
import matplotlib.pyplot as plt

# Define the data file path
file_path = 'files/output_sampled_ipv6.txt'

# Define the column names
columns = ['src', 'dst', 'pro', 'ok', 'sport', 'dport', 'packets',
```

```
'bytes', 'flows', 'first', 'latest']

# Read the file into a DataFrame
df = pd.read_csv(file_path, sep='\t', header=None, names=columns)

# Calculate the aggregated data volume for each user (source IP
address)
user_data_volume =
df.groupby('src')['bytes'].sum().sort_values(ascending=False)

plt.figure(figsize=(12, 6))
user_data_volume.plot(kind='bar', color='skyblue', logy=True)
plt.title('User Aggregated Data Volume (IPv6)')
plt.xlabel('User IP Address')
plt.ylabel('Aggregated Data Volume (bytes)')
plt.xticks(rotation=90, fontsize=0.0001)
plt.tight_layout()
plt.show()
```

The code's purpose is to process network traffic data stored in a file, specifically focusing on IPv6 traffic. It reads the data into a DataFrame, calculates the aggregated data volume for each user based on their source IP address, and then visualizes this aggregated data volume using a bar chart.





These two graphs respectively illustrate the aggregated data volume per user within IPv4 and IPv6 networks. Both charts exhibit a highly skewed distribution of data volumes among users. A small number of users account for a disproportionately large share of the data traffic, whereas the majority of users consume much less data. The data follows a long-tail distribution, which is typical in network traffic analysis. After an initial steep decline, there is a long tail representing a large number of users with low data volumes. Comparing IPv4 and IPv6, the overall shape of the distribution is similar, indicating that the usage patterns between the two protocols are not significantly different.

- Compare the results to the original task where you used your subnetwork (FS2) only. Can you say the characteristics of your subnetwork is representative? Is there a difference between IPv4 and IPv6?

  When comparing the traffic patterns sampled from the subnetwork (FS2) with the overall dataset (FS1), FS2 also reflects FS1's traffic distribution and variability. Therefore, I believe the characteristics of FS2 can represent the larger dataset.

  The difference between IPv4 and IPv6 traffic in FS1 suggests they may have different usage patterns or adoption rates. IPv4 shows more significant traffic volume peaks, which could be due to its wider adoption and the nature of the services it supports. Although IPv6 also shows changes in traffic volume, it may not have the same intensity or pattern of usage as IPv4, potentially due to its evolving adoption and different network service

characteristics.

- Conclusions

  - Traffic volume at different time scales. Are there any identifiable patterns or trends that you observed?

    In both the per-minute and per-hour traffic volume charts, identifiable peaks in traffic volume suggest the possibility of periodic high-usage events or scheduled activities. The hourly charts are smoother and show less variability because they aggregate traffic over a longer period, diluting the short-term peaks observed in the per-minute charts. For IPv6, traffic starts with a high volume and then quickly stabilizes, whereas IPv4 traffic displays more variability over time with several distinct peaks, indicating that IPv4 may be handling a mix of baseline traffic and bursty events.

  - Identify the top 5 most common applications by studying their port numbers.

    Flow data:

    1) Port 443: HTTPS (Hypertext Transfer Protocol Secure) - Port 443 is commonly used for secure web browsing, where data is encrypted between the user's web browser and the web server. It ensures secure online transactions and protects sensitive information.
    2) Port 80: HTTP (Hypertext Transfer Protocol) - Port 80 is used for standard, non-secure web browsing. It is the default port for web traffic and is used for accessing websites and web services.
    3) Port 5223: Apple Push Notification Service (APNS) - Port 5223 is associated with Apple's APNS, used for sending push notifications to iOS devices. It enables applications to deliver real-time updates to users.
    4) Port 3: Compression Process - Port 3 is not commonly associated with a specific application, and it's not a well-known port for standard services. It may be used for various purposes depending on the specific configuration.
    5) Port 993: IMAPS (Internet Message Access Protocol Secure) - Port 993 is used for secure email communication using IMAPS. It provides encrypted access to email messages stored on an email server, ensuring the privacy and security of email communications.

IPv4:

1) Port 23: Telnet - Telnet is a network protocol used for remote terminal connection and management of devices. It allows users to log in to a remote computer and execute commands.
2) Port 53: Domain Name System (DNS) - DNS is a fundamental network protocol used to translate human-readable domain names into IP addresses and vice versa. It is essential for internet browsing and communication.
3) Port 22: Secure Shell (SSH) - SSH is a secure network protocol used for secure remote access and control of network devices and servers. It provides encrypted communication and authentication.
4) Port 7547: CPE WAN Management Protocol (CWMP) - CWMP is a protocol used for managing customer premises equipment (CPE) in broadband networks, such as DSL modems and routers, by service providers.
5) Port 445: Microsoft-DS (Directory Services) - Port 445 is associated with the Microsoft-DS protocol used for file and printer sharing, as well as other network services, in Windows environments.

IPv6:

1) Port 53: Domain Name System (DNS) - DNS is a fundamental network protocol used for mapping domain names to IP addresses and vice versa. It is widely used on the internet for resolving domain names to access websites and other network resources.
2) Port 80: Hypertext Transfer Protocol (HTTP) - HTTP is a protocol used for transmitting text, images, videos, and other content over the web. It forms the basis for web browsing and accessing web pages.
3) Port 443: Secure Sockets Layer (SSL)/Transport Layer Security (TLS) - Port 443 is commonly used for the HTTPS protocol, which is the secure version of HTTP, ensuring encrypted web communication to maintain data confidentiality and integrity.
4) Port 10050: Zabbix Agent - Port 10050 is typically associated with the Zabbix monitoring system agent, used for monitoring and collecting performance data from servers and network devices.
5) Port 123: Network Time Protocol (NTP) - NTP is used for synchronizing the time of computers and other network devices. It ensures that the clocks of various devices

remain accurately synchronized.

- What kind of users there are in the network? Speculate on what kind of network this network could be based on traffic volumes and user profiles. Is your subnetwork different from larger population?

1) Types of Users in the Network:
   - ✓ Heavy Users: The presence of significant traffic peaks, especially in IPv4, suggests heavy users or power users. These could be servers, large-scale file-sharing systems, or streaming services consuming and distributing large amounts of data.
   - ✓ General Users: The long tail distribution observed indicates a large number of general users with relatively low data usage. This group likely includes regular consumers using the network for typical activities like browsing, email, and standard-definition video streaming.
   - ✓ Specialized Users: The use of specific ports (like 443, 80, 23, 22) suggests users engaged in activities like secure web browsing (HTTPS), remote server access (SSH, Telnet), and DNS services.

2) Nature of the Network:
   - ✓ The mix of heavy and general users, along with the types of services inferred from port usage, suggests a diverse network. This could be a residential or commercial ISP network, a large corporate network, or a university campus network.
   - ✓ The presence of significant volumes of HTTPS and HTTP traffic points to regular web browsing as a primary activity, common in commercial and educational networks.
   - ✓ The usage patterns of ports like 22 (SSH) and 23 (Telnet) might indicate a network with administrative or developmental activities, perhaps with a number of servers or development machines.

3) Difference from larger population
   - ✓ My subnet displays a sharp drop in data volume after the top-ranking users, which may indicate a network with a few heavy users or servers and many light users. In contrast, the larger population should show a more gradual decline,

indicating a more uniform distribution of data usage among users.

- ✓ In my subnet, data volume noticeably leans toward the top IP addresses, suggesting that this might be a corporate or institutional network with specific nodes handling significant data loads. In a larger network, a smoother gradient might be observed, indicating a network without such pronounced outliers in terms of data usage, such as a residential ISP where individual households have a more uniform data usage.
- ✓ The presence of very high-volume IP addresses in my subnet might indicate specialized activities like hosting services, large file transfers, or data processing tasks, which could be characteristic of service providers or large organizations with centralized data operations. In contrast, the larger network's more uniform usage suggests that it might not have as many specialized high-demand nodes and might be structured to provide more consistent service across all nodes, such as a smaller business, educational institution, or community network.

- Comparison of the above results with the result from data set PS2.

1) Traffic Volume and Patterns: Personal networks typically exhibit simpler overall traffic patterns with short-duration traffic peaks and smaller data volumes, primarily reflecting individual internet usage habits, such as evening or weekend browsing, video watching, or social media interaction. In contrast, public networks, like those within companies or schools, due to a larger number of users and devices, display higher usage frequency and larger data volumes. These networks experience more frequent and longer-lasting traffic peaks, reflecting a combination of diverse work and leisure activities.

2) User Behavior Characteristics: In personal networks, user behavior tends to be more uniform, primarily revolving around daily internet activities such as online shopping, news browsing, or using streaming services, which usually don't require substantial data transfer. On the other hand, in public network environments, the variety of user behaviors is more extensive, ranging from basic web browsing to complex data processing and online collaboration. For instance, corporate networks might encompass large-scale file transfers, remote conferencing, complex data analysis, and cloud computing services. This diversity results in more varied and complex

data usage volumes and types in public networks compared to personal networks.

3) Service and Application Diversity: Users of private networks typically access a limited set of services that meet basic personal needs, like personal email, social media platforms, and online entertainment content. While these services are widespread, their data usage in terms of volume and type is relatively limited. In contrast, public networks might need to support a broader range of services to accommodate the needs of various users and operations. These might include enterprise resource management systems, cloud storage and computing services, professional online collaboration tools, and access to external websites. The usage of these services not only brings higher data traffic to the network but also imposes higher demands in terms of security and reliability. This diversity makes the maintenance and management of public networks more complex than that of private networks.

Aalto University
School of Electrical
Engineering

Task 3: Analysing active measurements

- Latency measurements (data sets AS1.x), where x includes:

  AS1.d1: Name server1 with DNS: dns-st.bahnhof.net.txt

  AS1.d2: Name server2 with DNS: ns1.bahnhof.net.txt

  AS1.d3: Name server3 with DNS: southeast-2.dns-au.st.txt

  AS1.n1: Name server1 with ICMP: dns-st.bahnhof.net.txt

  AS1.n2: Name server2 with ICMP: ns1.bahnhof.net.txt

  AS1.n3: Name server3 with ICMP: southeast-2.dns-au.st.txt

  AS1.r1: Research server1: cbg-uk.ark.caida.org.txt

  AS1.r2: Research server2: bjl-gm.ark.caida.org.txt

  AS1.r3: Research server3: msy-us.ark.caida.org.txt

  AS1.i1: Iperf server1: ok1.iperf.comnet-student.eu.txt

  AS1.i2: Iperf server2: blr1.iperf.comnet-student.eu.txt

- Throughput measurements (data sets AS2.x), where x includes:

  AS2.i1: Iperf server1: ok1.iperf.comnet-student.eu.txt

  AS2.i2: Iperf server2: blr1.iperf.comnet-student.eu.txt

Sample:

```
;; Query time: 12775 usec
;CURL: 16.092021 1.031525 0.999642
;; Query time: 12377 usec
;CURL: 2.925309 2.846126 2.819373
;; Query time: 12715 usec
;CURL: 1.841303 1.780881 1.753875
;; Query time: 12331 usec
;CURL: 2.118491 2.044278 2.018138
;; Query time: 2531210 usec
;CURL: 10.418708 2.821009 2.790507
```

```
;; Query time: 13276 usec
;CURL: 16.100855 1.034431 1.000153
;; Query time: 13395 usec
;CURL: 2.894204 2.840337 2.819398
;; Query time: 13155 usec
;CURL: 1.836068 1.776338 1.753678
;; Query time: 15177 usec
;CURL: 2.094822 2.038630 2.018427
;; Query time: 2496893 usec
;CURL: 10.419783 2.822090 2.786697
```

```
;; Query time: 322285 usec
;CURL: 16.410200 1.044784 0.999718
;; Query time: 306800 usec
;CURL: 3.191081 2.841170 2.819277
;; Query time: 307191 usec
;CURL: 2.124918 1.780938 1.753953
;; Query time: 307074 usec
;CURL: 2.392769 2.043346 2.018237
;; Query time: 2855592 usec
;CURL: 10.717499 2.818339 2.787209
```

```
--- dns-st.bahnhof.net ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 14.970/15.667/17.279/0.846 ms
PING dns-st.bahnhof.net (79.136.119.20) 56(84) bytes of data.
[1695812045.260879] 64 bytes from dns-st.bahnhof.net (79.136.119.20): icmp_seq=1 ttl=55 time=15.6 ms
[1695812045.276599] 64 bytes from dns-st.bahnhof.net (79.136.119.20): icmp_seq=2 ttl=55 time=15.0 ms
[1695812046.278369] 64 bytes from dns-st.bahnhof.net (79.136.119.20): icmp_seq=3 ttl=55 time=15.1 ms
[1695812047.278820] 64 bytes from dns-st.bahnhof.net (79.136.119.20): icmp_seq=4 ttl=55 time=13.8 ms
[1695812048.281370] 64 bytes from dns-st.bahnhof.net (79.136.119.20): icmp_seq=5 ttl=55 time=14.9 ms
```

```
--- ns1.bahnhof.net ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 14.916/19.042/31.577/6.423 ms
PING ns1.bahnhof.net (195.178.160.2) 56(84) bytes of data.
[1695812045.559724] 64 bytes from ns1.bahnhof.net (195.178.160.2): icmp_seq=1 ttl=54 time=15.3 ms
[1695812045.574838] 64 bytes from ns1.bahnhof.net (195.178.160.2): icmp_seq=2 ttl=54 time=14.5 ms
[1695812046.577405] 64 bytes from ns1.bahnhof.net (195.178.160.2): icmp_seq=3 ttl=54 time=15.1 ms
[1695812047.580017] 64 bytes from ns1.bahnhof.net (195.178.160.2): icmp_seq=4 ttl=54 time=15.8 ms
[1695812048.581225] 64 bytes from ns1.bahnhof.net (195.178.160.2): icmp_seq=5 ttl=54 time=14.8 ms
```

```
--- southeast-2.dns-au.st ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 326.350/329.090/339.193/5.055 ms
PING southeast-2.dns-au.st (52.63.82.137) 56(84) bytes of data.
[1695812044.961934] 64 bytes from ec2-52-63-82-137.ap-southeast-2.compute.amazonaws.com (52.63.82.137): icmp_seq=1 ttl=35 time=325 ms
[1695812045.287546] 64 bytes from ec2-52-63-82-137.ap-southeast-2.compute.amazonaws.com (52.63.82.137): icmp_seq=2 ttl=35 time=325 ms
[1695812046.288696] 64 bytes from ec2-52-63-82-137.ap-southeast-2.compute.amazonaws.com (52.63.82.137): icmp_seq=3 ttl=35 time=326 ms
[1695812047.286908] 64 bytes from ec2-52-63-82-137.ap-southeast-2.compute.amazonaws.com (52.63.82.137): icmp_seq=4 ttl=35 time=324 ms
[1695812048.295828] 64 bytes from ec2-52-63-82-137.ap-southeast-2.compute.amazonaws.com (52.63.82.137): icmp_seq=5 ttl=35 time=332 ms
```

```
--- cbg-uk.ark.caida.org ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 35.853/36.450/37.014/0.491 ms
PING cbg-uk.ark.caida.org (128.232.97.9) 56(84) bytes of data.
[1695808442.391055] 64 bytes from monitor.ark.caida.cl.cam.ac.uk (128.232.97.9): icmp_seq=1 ttl=47 time=49.3 ms
[1695808442.570048] 64 bytes from monitor.ark.caida.cl.cam.ac.uk (128.232.97.9): icmp_seq=2 ttl=47 time=39.2 ms
[1695808443.568067] 64 bytes from monitor.ark.caida.cl.cam.ac.uk (128.232.97.9): icmp_seq=3 ttl=47 time=35.9 ms
[1695808444.570459] 64 bytes from monitor.ark.caida.cl.cam.ac.uk (128.232.97.9): icmp_seq=4 ttl=47 time=36.7 ms
[1695808445.572672] 64 bytes from monitor.ark.caida.cl.cam.ac.uk (128.232.97.9): icmp_seq=5 ttl=47 time=37.7 ms
```

```
--- blr1.iperf.comnet-student.eu ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4633ms
rtt min/avg/max/mdev = 344.979/346.394/347.890/1.130 ms
PING blr1.iperf.comnet-student.eu (142.93.213.224) 56(84) bytes of data.
[1695808444.009265] 64 bytes from 142.93.213.224 (142.93.213.224): icmp_seq=1 ttl=49 time=340 ms
[1695808444.356839] 64 bytes from 142.93.213.224 (142.93.213.224): icmp_seq=2 ttl=49 time=347 ms
[1695808445.349339] 64 bytes from 142.93.213.224 (142.93.213.224): icmp_seq=3 ttl=49 time=338 ms
[1695808446.351641] 64 bytes from 142.93.213.224 (142.93.213.224): icmp_seq=4 ttl=49 time=338 ms
[1695808447.352613] 64 bytes from 142.93.213.224 (142.93.213.224): icmp_seq=5 ttl=49 time=338 ms
```

```
--- msy-us.ark.caida.org ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 163.931/164.860/166.247/0.833 ms
PING msy-us.ark.caida.org (98.164.118.12) 56(84) bytes of data.
[1695808442.531334] 64 bytes from ip98-164-118-12.no.no.cox.net (98.164.118.12): icmp_seq=1 ttl=45 time=166 ms
[1695808443.131246] 64 bytes from ip98-164-118-12.no.no.cox.net (98.164.118.12): icmp_seq=2 ttl=45 time=165 ms
[1695808444.133598] 64 bytes from ip98-164-118-12.no.no.cox.net (98.164.118.12): icmp_seq=3 ttl=45 time=167 ms
[1695808445.132556] 64 bytes from ip98-164-118-12.no.no.cox.net (98.164.118.12): icmp_seq=4 ttl=45 time=165 ms
[1695808446.132695] 64 bytes from ip98-164-118-12.no.no.cox.net (98.164.118.12): icmp_seq=5 ttl=45 time=165 ms
```

```
iperf Done.
2023-09-29 21:21:31
Connecting to host ok1.iperf.comnet-student.eu, port 5201
[  5] local 172.17.175.205 port 41654 connected to 195.148.124.36 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  56.1 MBytes   471 Mbits/sec    0   2.73 MBytes
[  5]   1.00-2.00   sec  52.5 MBytes   440 Mbits/sec    0   3.88 MBytes
[  5]   2.00-3.00   sec  63.8 MBytes   535 Mbits/sec    0   3.88 MBytes
[  5]   3.00-4.00   sec  73.8 MBytes   619 Mbits/sec    0   3.88 MBytes
[  5]   4.00-5.00   sec  71.2 MBytes   598 Mbits/sec    0   3.88 MBytes
[  5]   5.00-6.00   sec  65.0 MBytes   545 Mbits/sec    0   3.88 MBytes
[  5]   6.00-7.00   sec  65.0 MBytes   545 Mbits/sec    0   3.88 MBytes
[  5]   7.00-8.00   sec  78.8 MBytes   661 Mbits/sec    0   3.88 MBytes
[  5]   8.00-9.00   sec  63.8 MBytes   535 Mbits/sec    0   3.88 MBytes
[  5]   9.00-10.00  sec  65.0 MBytes   545 Mbits/sec    0   3.88 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   655 MBytes   549 Mbits/sec    0             sender
[  5]   0.00-10.02  sec   655 MBytes   548 Mbits/sec                  receiver
```

```
iperf Done.
2023-09-30 22:54:01
Connecting to host blr1.iperf.comnet-student.eu, port 5202
[  5] local 172.17.175.205 port 34874 connected to 142.93.213.224 port 5202
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   257 KBytes  2.11 Mbits/sec    0   56.6 KBytes
[  5]   1.00-2.00   sec  1.93 MBytes  16.2 Mbits/sec    0    436 KBytes
[  5]   2.00-3.00   sec  2.26 MBytes  19.0 Mbits/sec    1    478 KBytes
[  5]   3.00-4.00   sec  2.50 MBytes  21.0 Mbits/sec    0    544 KBytes
[  5]   4.00-5.00   sec  1.25 MBytes  10.5 Mbits/sec    0    592 KBytes
[  5]   5.00-6.00   sec  1.25 MBytes  10.5 Mbits/sec    1    431 KBytes
[  5]   6.00-7.00   sec  1.25 MBytes  10.5 Mbits/sec    0    472 KBytes
[  5]   7.00-8.00   sec  1.25 MBytes  10.5 Mbits/sec    0    496 KBytes
[  5]   8.00-9.00   sec  1.25 MBytes  10.5 Mbits/sec    1    358 KBytes
[  5]   9.00-10.00  sec  1.25 MBytes  10.5 Mbits/sec    0    393 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  14.4 MBytes  12.1 Mbits/sec    3             sender
[  5]   0.00-10.33  sec  12.0 MBytes  9.71 Mbits/sec                  receiver
```

1. Latency data plots (AS1.x)

   - Excludes packet loss

```
# Initialize a list to store extracted latency values
latency_values = []

# Open the file and read each line
file_path = 'files/ping/southeast-2.dns-au.st.txt'
with open(file_path, 'r') as file:
    for line in file:
        if 'time=' in line:
            # Find lines containing 'time=' and split the string
            parts = line.split()
            for part in parts:
                if part.startswith('time='):
                    # Extract the latency value and remove 'ms'
                    latency = part.split('=')[1].rstrip(' ms')
                    try:
                        # Convert the extracted latency value to a
floating-point number and store it
                        latency_values.append(float(latency))
                    except ValueError:
                        # If conversion fails, skip the value
                        continue

latency_values.sort()
print(latency_values)

import matplotlib.pyplot as plt

# Create a boxplot, assuming latency_values contains latency
measurements
fig, ax = plt.subplots(figsize=(10, 6))
boxplot_dict = ax.boxplot(latency_values, patch_artist=True)

# Annotate medians
medians = [median.get_ydata()[0] for median in
boxplot_dict['medians']]
for tick, median in zip(ax.get_xticks(), medians):
```

```
    ax.text(tick, median, f'Median: {median:.2f}', ha='center',
va='center', fontdict={'fontsize': 8, 'color': 'white'})

# Annotate quartiles
boxes = [box.get_path().vertices for box in boxplot_dict['boxes']]
for box in boxes:
    box_bottom = box[0, 1]
    box_top = box[2, 1]
    ax.text(box[0, 0], box_bottom, f'Q1: {box_bottom:.2f}',
ha='center', va='top', fontdict={'fontsize': 8})
    ax.text(box[2, 0], box_top, f'Q3: {box_top:.2f}', ha='center',
va='bottom', fontdict={'fontsize': 8})

# Set title and labels
ax.set_title(f'Latency Measurements Box Plot {file_path.split("/")[-
1]}')
ax.set_ylabel('Latency (ms)')
ax.set_xlabel('Measurements')
plt.grid(True)

# Show the plot
plt.show()
```

This code reads latency measurements from a file, extracts the latency values, and creates a box plot to visualize the distribution of latency measurements. The latency values are sorted and then displayed in a box plot, including median values and quartiles. The resulting plot provides insights into the latency characteristics of the network or system being monitored.

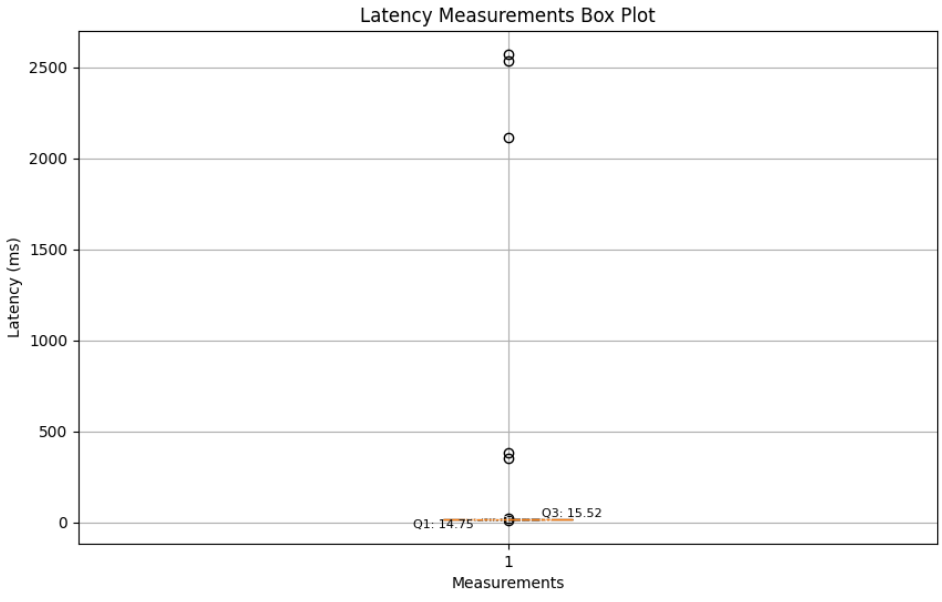
```
import matplotlib.pyplot as plt

# Initialize lists to store query time lines and CURL lines
query_time_lines = []
curl_lines = []

# Open the file and process each line
with open('files/dig/ns1.bahnhof.net.txt', 'r') as file:
    skip_next_line = False
    for line in file:
        stripped_line = line.strip()
```

```
        if skip_next_line:
            skip_next_line = False
            continue

        if "Return" in stripped_line:
            skip_next_line = True
        elif "Query" in stripped_line:
            query_time_lines.append(stripped_line)
        elif "CURL" in stripped_line:
            curl_lines.append(stripped_line)

# Extract query times and connect times
query_times_ms = [float(line.split()[3]) for line in
query_time_lines]
connect_times_ms = [float(line.split()[3]) for line in curl_lines]

# Calculate delays and sort them
delays_ms = [(query_time / 1000) + connect_time for query_time,
connect_time in zip(query_times_ms, connect_times_ms)]
delays_ms.sort()
print(delays_ms)

# Create a boxplot using delays_ms
fig, ax = plt.subplots(figsize=(10, 6))
boxplot_dict = ax.boxplot(delays_ms, patch_artist=True)

# Annotate median values
medians = [median.get_ydata()[0] for median in
boxplot_dict['medians']]
for tick, median in zip(ax.get_xticks(), medians):
    ax.text(tick, median, f'Median: {median:.2f}', ha='center',
va='center', fontdict={'fontsize': 8, 'color': 'white'})

# Annotate quartiles
boxes = [box.get_path().vertices for box in boxplot_dict['boxes']]
for box in boxes:
    box_bottom = box[0, 1]
    box_top = box[2, 1]
    ax.text(box[0, 0], box_bottom, f'Q1: {box_bottom:.2f}',
```

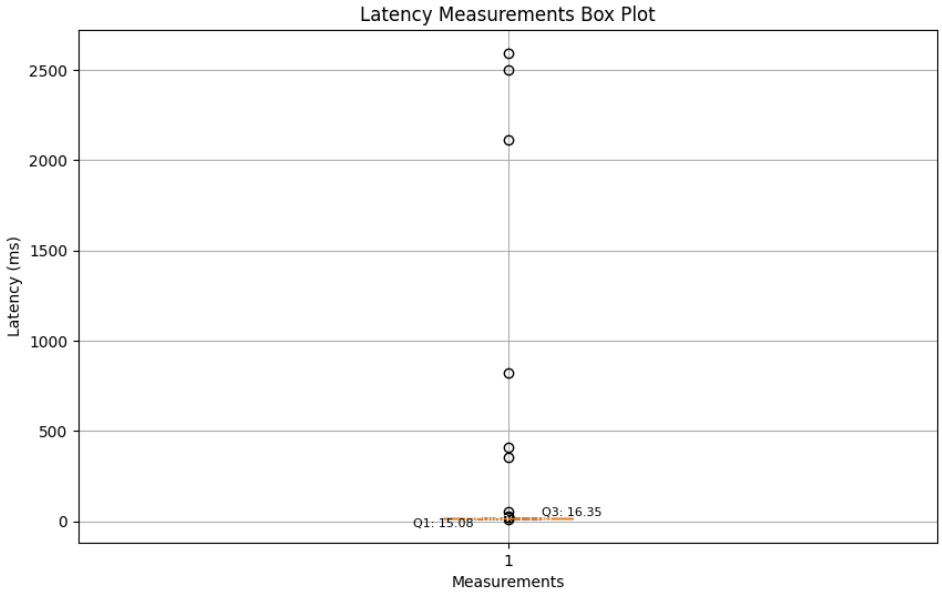
```
ha='center', va='top', fontdict={'fontsize': 8})
    ax.text(box[2, 0], box_top, f'Q3: {box_top:.2f}', ha='center',
va='bottom', fontdict={'fontsize': 8})

# Set title and labels
ax.set_title('Latency Measurements Box Plot')
ax.set_ylabel('Latency (ms)')
ax.set_xlabel('Measurements')
plt.grid(True)
# Show the plot
plt.show()
```

The functionality of this code is to analyze a network log file containing query times and connect times. It begins by initializing lists to store lines containing query time and CURL time. The code reads each line from the file, identifies and extracts relevant query and CURL lines while skipping unnecessary lines marked by "Return." The extracted query and CURL times are collected and converted into lists. The code then calculates delays by adding query times (converted to seconds) and connect times, sorts these delay values in ascending order, and displays them in a box plot using Matplotlib. The box plot visualizes the distribution of delay data, including median and quartiles, aiding in the analysis of network performance or system response times.

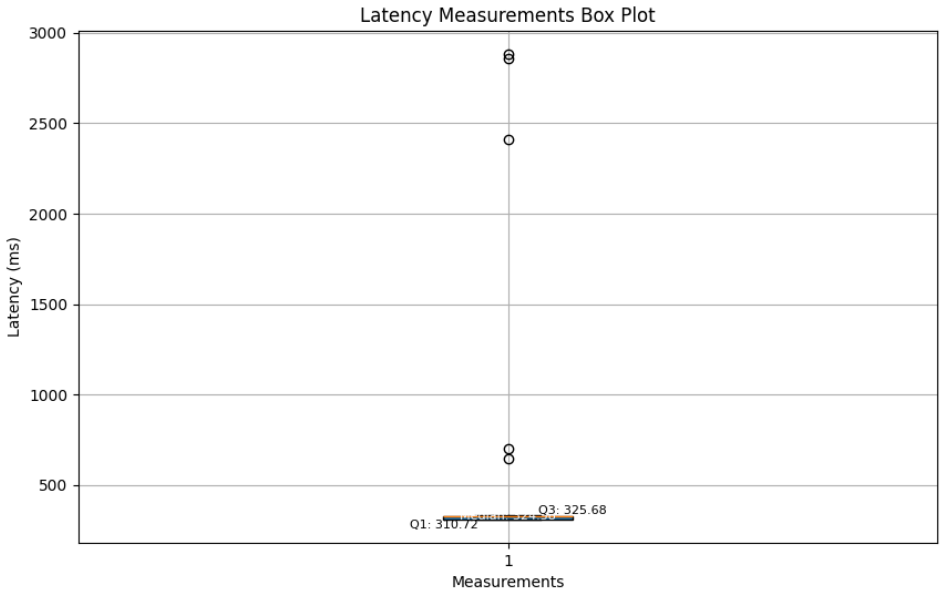
AS1.d1:



AS1.d2:

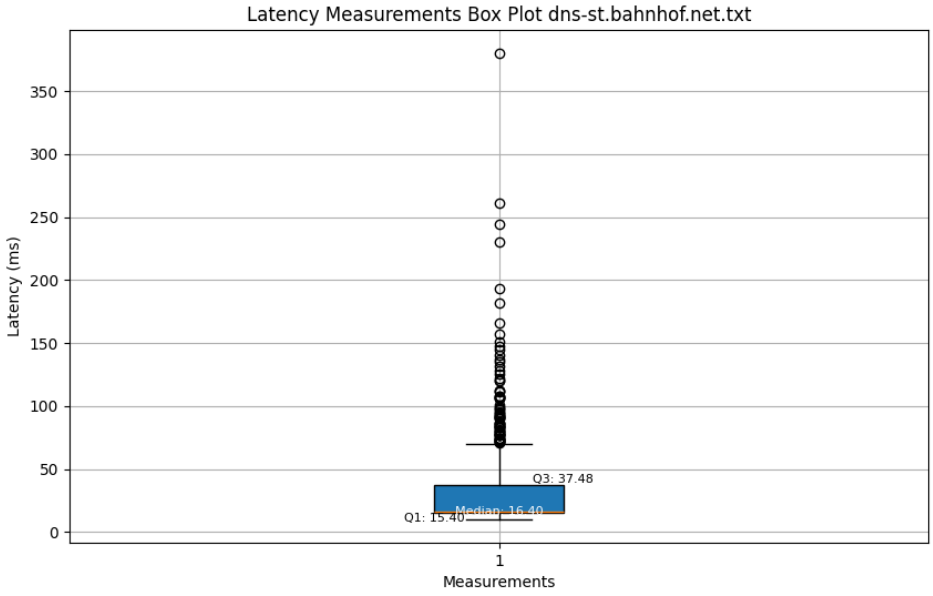

Latency Measurements Box Plot
Latency (ms)
2500
2000
1500
1000
500
0
Q3: 16.35
Q1: 15.08
1
Measurements

AS1.d3:

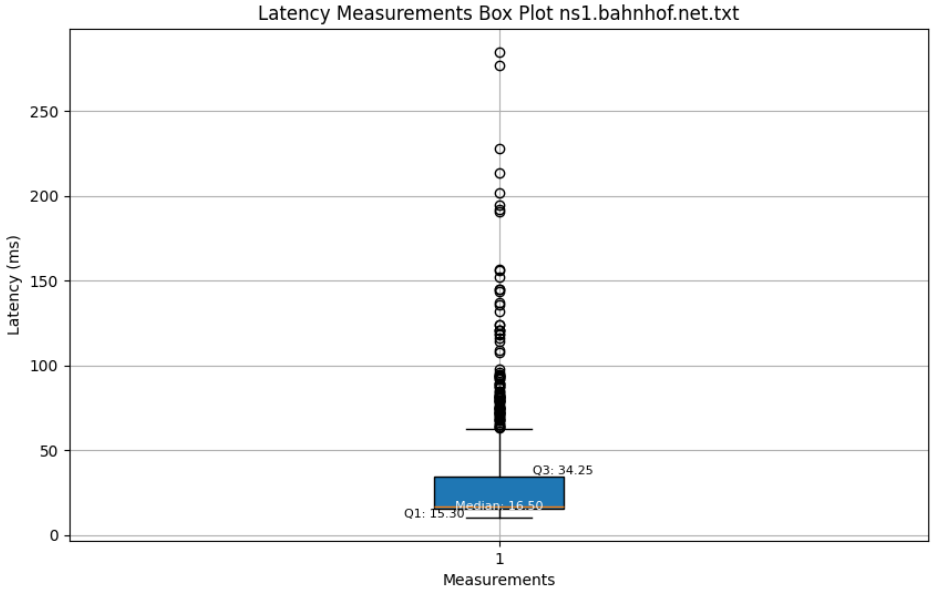

Latency Measurements Box Plot
Latency (ms)
3000
2500
2000
1500
1000
500
Q3: 325.68
Q1: 310.72
1
Measurements

Aalto University
School of Electrical
Engineering

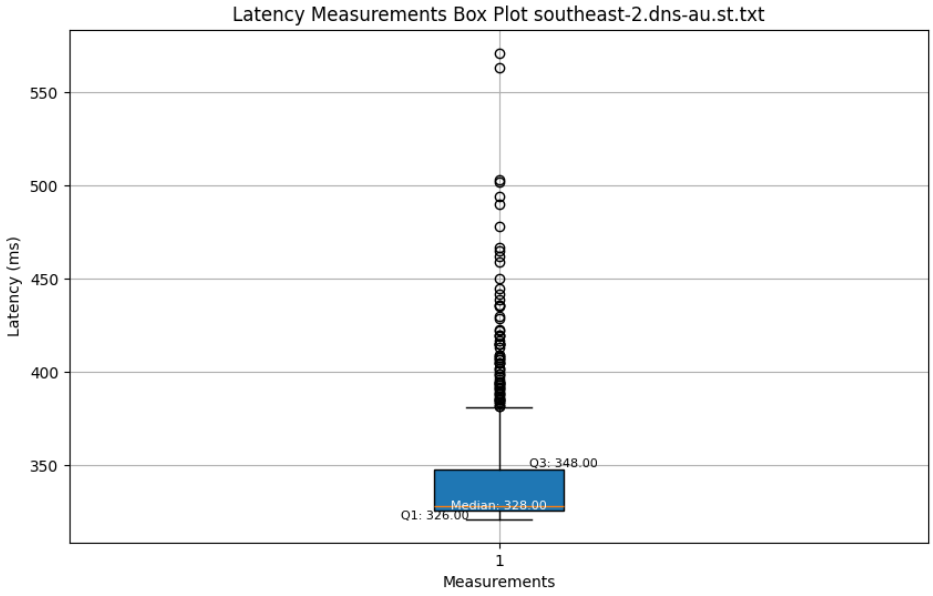
AS1.n1:



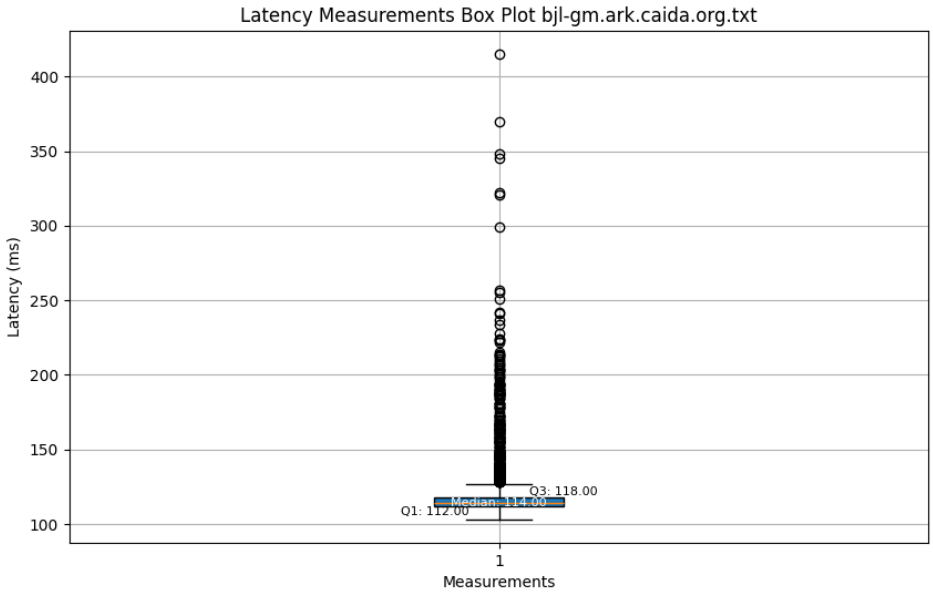
AS1.n2:



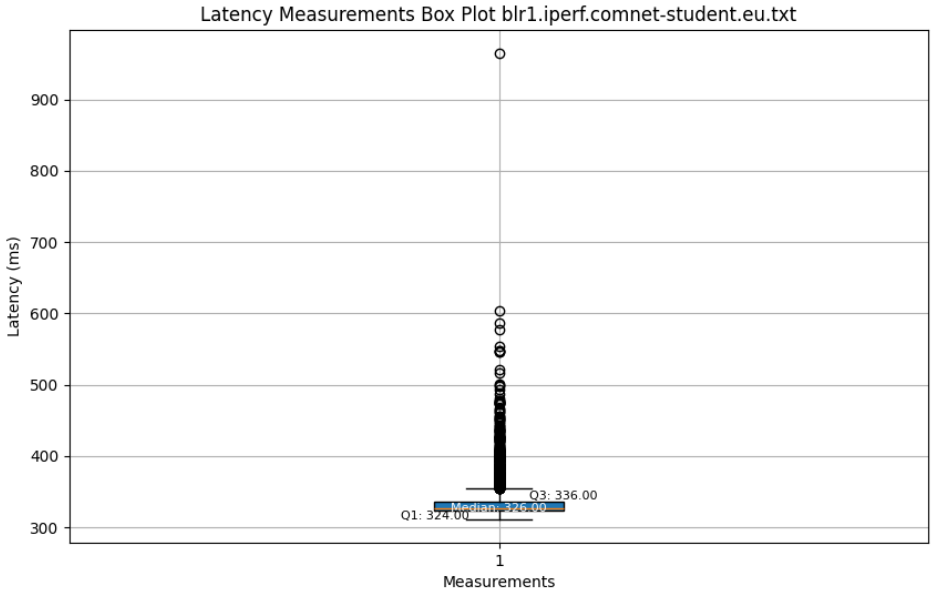
AS1.n3:


Latency Measurements Box Plot southeast-2.dns-au.st.txt
Latency (ms)
550
500
450
400
350
Q3: 348.00
Median: 328.00
Q1: 326.00
1
Measurements

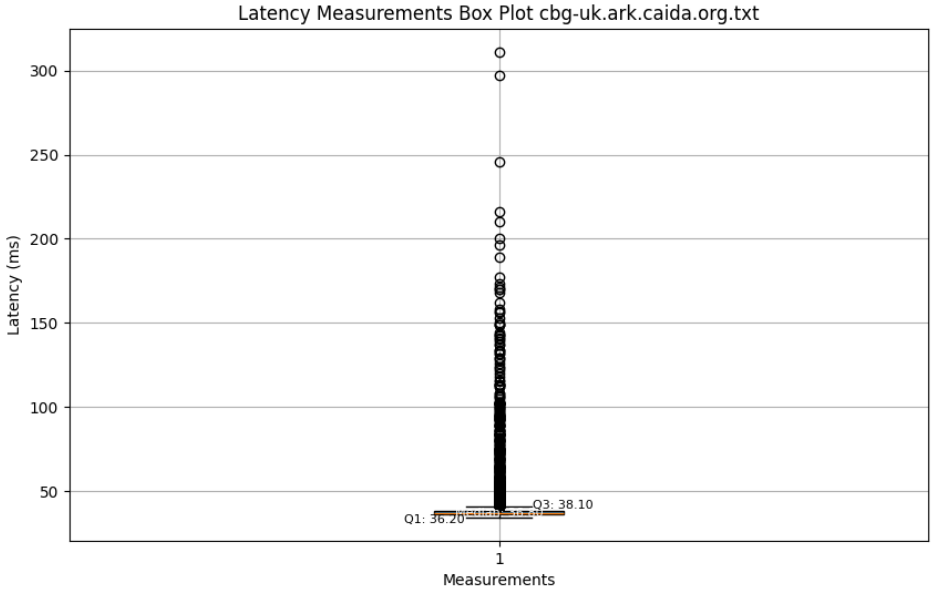
AS1.r1:


Latency Measurements Box Plot bjl-gm.ark.caida.org.txt
Latency (ms)
400
350
300
250
200
150
100
Q3: 118.00
Median: 114.00
Q1: 112.00
1
Measurements

AS1.r2:

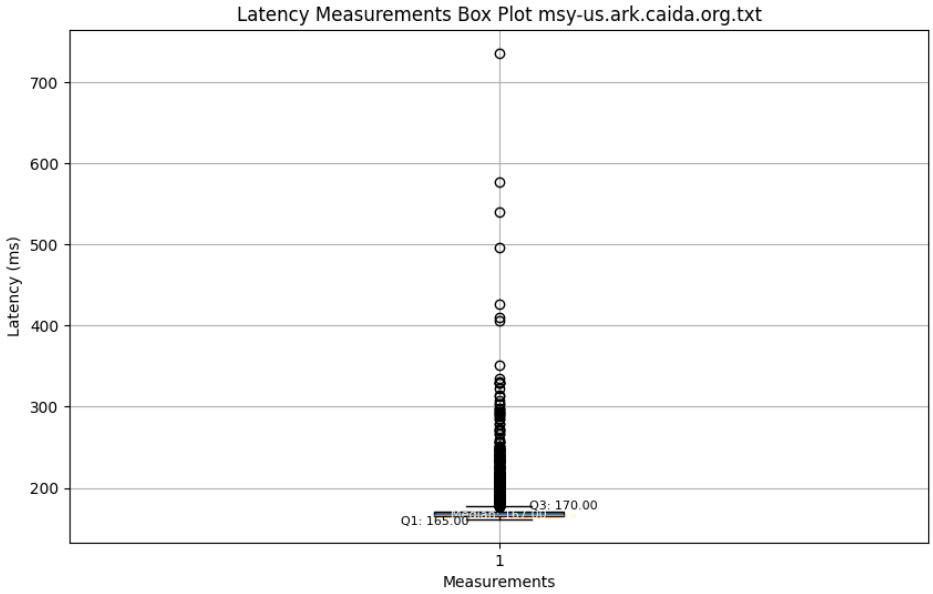

Latency Measurements Box Plot blr1.iperf.comnet-student.eu.txt
Latency (ms)
900
800
700
600
500
400
300
Q3: 336.00
Median: 326.00
Q1: 324.00
1
Measurements

AS1.r3:

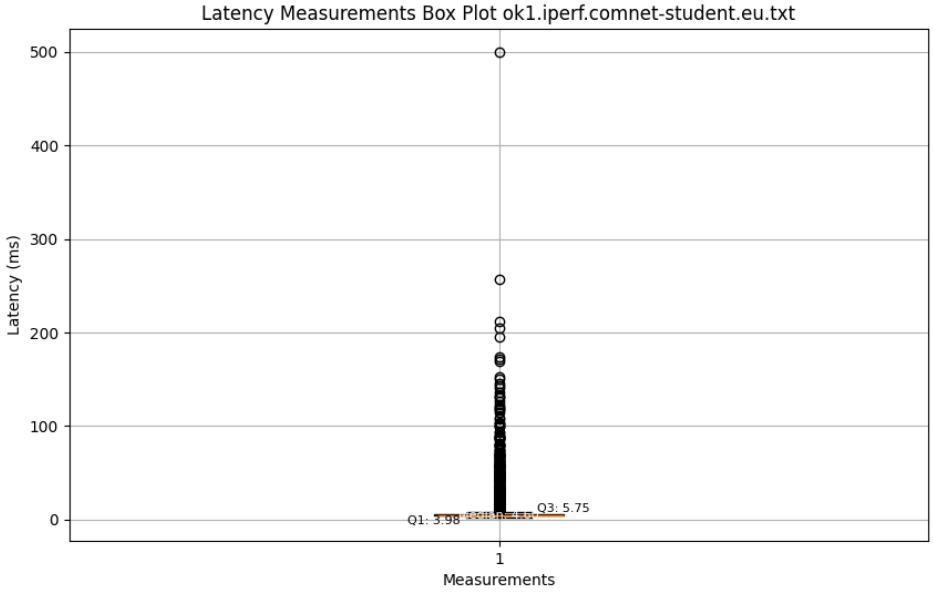

Latency Measurements Box Plot cbg-uk.ark.caida.org.txt
Latency (ms)
300
250
200
150
100
50
Q3: 38.10
Q1: 36.20
1
Measurements

Aalto University
School of Electrical
Engineering

AS1.i1:



AS1.i2:



- Notable observations

AS1.d1: The median latency is approximately 15 milliseconds, indicating that most measurement values are relatively low. The interquartile range is very small, suggesting that the majority of the data is concentrated within a narrow latency range. There are some outliers well above Q3, representing individual latency spikes.

AS1.d2: The median latency is around 16 milliseconds. The interquartile range is slightly larger, indicating some dispersion in the data. There are many outliers well above Q3, suggesting frequent latency spikes.

AS1.d3: The median latency is approximately 325 milliseconds, indicating a higher latency level. The interquartile range is quite large, reflecting significant differences among latency measurements. There are several latency outliers, with values much higher than the rest of the measurements.

AS1.n1: The median latency is around 16 milliseconds, indicating that most ICMP response times are fast. The interquartile range is very tight, indicating high consistency among the measurement results. There are a few outliers, indicating occasional latency spikes.

AS1.n2: The median latency is also around 16 milliseconds. The interquartile range is slightly wider than AS1.n1, but the overall measurements are still concentrated. There are some outliers, suggesting occasional latency spikes.

AS1.n3: The median latency is approximately 328 milliseconds, indicating a generally higher latency level. The interquartile range is relatively consistent with AS1.n2, suggesting data consistency. There are few outliers, indicating infrequent occurrences of high latency in ICMP requests.

AS1.r1: The median latency is approximately 38 milliseconds. The interquartile range (IQR), which represents the middle 50% of the data, is very narrow, indicating that most of the latency measurements are closely clustered around the median. There are many outliers extending to about 250 milliseconds, suggesting occasional spikes in latency.

AS1.r2: The median latency is approximately 170 milliseconds. The IQR is wider than that of server 1, indicating greater variability in the latency measurements. There is a significant number of

outliers exceeding 400 milliseconds, showing that this server occasionally experiences high latency.

AS1.r3: The median latency is approximately 170 milliseconds. The IQR is quite small, suggesting less variability in the data compared to server 2. There are outliers, with some latency measurements exceeding 600 milliseconds.

AS1.i1: The median latency is very low, about 5.75 milliseconds, which is the lowest among all the servers, including the research servers. The IQR is extremely tight, indicating a highly consistent latency performance. The range of outliers is minimal, with the highest latency measurement just slightly above 100 milliseconds.

AS1.i2: The median latency of this server is the highest among all the servers, at approximately 336 milliseconds. The IQR is relatively large, showing more variability in latency compared to the other iperf server. The range of outliers is wide, extending to nearly 900 milliseconds, suggesting that this server has the most latency spikes.

- Differences in AS1.d_N and AS1.n_N

Datasets AS1.d1, d2, d3: These datasets exhibit a very wide range of latency with a very high upper limit, indicating the potential for significant delays. The median values are relatively low compared to the overall range, suggesting that despite instances of extremely high latency, typical (median) performance is much better. The interquartile ranges (IQRs) are small, especially when compared to the range of outliers, indicating that the majority of data points are clustered within a narrow range, but with notable spikes. All three datasets have outliers, with some extreme cases where latency reaches up to 2000-3000 milliseconds, signaling occasional but very high latency peaks.

Datasets AS1.n1, n2, n3: Compared to the first three datasets, these datasets have a much narrower range of latency. This suggests a more stable and consistent latency experience. The medians are positioned towards the lower end of the range, indicating that more than half of the measurements are on the faster side. The IQRs are relatively tight, which shows that the middle 50% of measurements are closely grouped together. There are fewer and less extreme outliers compared to the first three datasets, indicating fewer instances of very high latency.

In summary, the AS1.d datasets indicate the possibility of high latency and more variability, while the AS1.n datasets display a more stable latency performance with fewer extreme outliers.

- Includes packet loss

```
import matplotlib.pyplot as plt

# Initialize a list to store latency measurements
latency_measurements = []
file_path = 'files/ping/bjl-gm.ark.caida.org.txt'

# Open the file and read each line
with open(file_path, 'r') as file:
    for line in file:
        if 'time=' in line:
            # Find lines containing 'time=' and split the string
            parts = line.split()
            for part in parts:
                if part.startswith('time='):
                    # Extract the latency measurement and remove 'ms'
                    latency_value = part.split('=')[1].rstrip(' ms')
                    try:
                        # Convert the latency measurement to a float and
store it
                        if float(latency_value) > 2000.0:
                            latency_measurements.append(2000.0)
                        else:

latency_measurements.append(float(latency_value))
                    except ValueError:
                        # If conversion fails, skip the value
                        continue
        if 'packet loss' in line:
            parts = line.split(',')
            for part in parts:
                if 'packet loss' in part:
                    # Extract the numeric value before the percentage
sign
                    packet_loss_percentage = part.split('%')[0].strip()
                    try:
                        # Convert the packet loss percentage to a float
and add outliers to the list
                        for i in range(0,
```

```
int(float(packet_loss_percentage) / 20.0)):
                        latency_measurements.append(2000.0)
                except ValueError:
                    # If conversion fails, ignore the value
                    continue

# Create a boxplot using latency_measurements
fig, ax = plt.subplots(figsize=(10, 6))
boxplot_dict = ax.boxplot(latency_measurements, patch_artist=True)

# Annotate median values
medians = [median.get_ydata()[0] for median in
boxplot_dict['medians']]
for tick, median in zip(ax.get_xticks(), medians):
    ax.text(tick, median, f'Median: {median:.2f}', ha='center',
va='center', fontdict={'fontsize': 8, 'color': 'white'})

# Annotate quartile values
boxes = [box.get_path().vertices for box in boxplot_dict['boxes']]
for box in boxes:
    box_bottom = box[0, 1]
    box_top = box[2, 1]
    ax.text(box[0, 0], box_bottom, f'Q1: {box_bottom:.2f}',
ha='center', va='top', fontdict={'fontsize': 8})
    ax.text(box[2, 0], box_top, f'Q3: {box_top:.2f}', ha='center',
va='bottom', fontdict={'fontsize': 8})

# Set title and labels
ax.set_title('Latency Measurements Box Plot')
ax.set_ylabel('Latency (ms)')
ax.set_xlabel('Measurements')
plt.grid(True)
plt.show()
```

This code reads latency measurements and packet loss information from a text file, processes the data to calculate latency values, and then creates a box plot visualization of these latency measurements. The latency measurements are extracted from lines containing 'time=' and 'packet loss' in the input file. Latency values are collected and outliers are added for packet loss percentages. The resulting latency measurements are visualized as a box plot, providing insights into the

distribution of latency values in the data. The box plot includes median and quartile annotations to summarize the statistics, and it is displayed using Matplotlib.

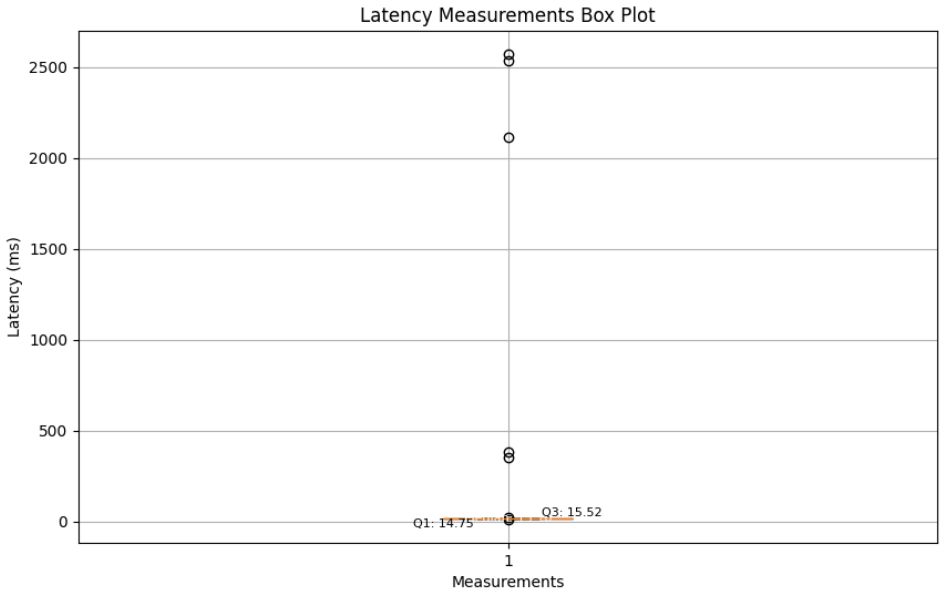
```
import re
import matplotlib.pyplot as plt

# Initialize lists with more descriptive names
query_time_lines = []
curl_lines = []
latency_ms = []

# Open and read the file
with open('files/dig/dns-st.bahnhof.net.reloaded.txt', 'r') as file:
    skip_next_line = False
    for line in file:
        stripped_line = line.strip()
        if skip_next_line:
            skip_next_line = False
            continue
        if re.search("Return", stripped_line):
            skip_next_line = True
            latency_ms.append(2000.0)
        elif re.search("Query", stripped_line):
            query_time_lines.append(stripped_line)
        elif re.search("CURL", stripped_line):
            curl_lines.append(stripped_line)

# Initialize lists for query time and connect time
query_times_ms = []
connect_times_ms = []

# Extract query and connect times
for item in query_time_lines:
    item_split = item.split(" ")
    query_time = float(item_split[3])
    query_times_ms.append(query_time)

for item in curl_lines:
    item_split = item.split(" ")
    connect_time = float(item_split[3])
```

```
    connect_times_ms.append(connect_time)

# Calculate total delay
total_delays_ms = [query_time / 1000 + connect_time for query_time,
connect_time in zip(query_times_ms, connect_times_ms)]
total_delays_ms.sort()

# Create a box plot
fig, ax = plt.subplots(figsize=(10, 6))
boxplot_dict = ax.boxplot(total_delays_ms, patch_artist=True)

# Annotate median
medians = [median.get_ydata()[0] for median in
boxplot_dict['medians']]
for tick, median in zip(ax.get_xticks(), medians):
    ax.text(tick, median, f'Median: {median:.2f}', ha='center',
va='center', fontdict={'fontsize': 8, 'color': 'white'})

# Annotate quartiles
boxes = [box.get_path().vertices for box in boxplot_dict['boxes']]
for box in boxes:
    box_bottom = box[0, 1]
    box_top = box[2, 1]
    ax.text(box[0, 0], box_bottom, f'Q1: {box_bottom:.2f}',
ha='center', va='top', fontdict={'fontsize': 8})
    ax.text(box[2, 0], box_top, f'Q3: {box_top:.2f}', ha='center',
va='bottom', fontdict={'fontsize': 8})

# Set title and labels
ax.set_title('Latency Measurements Box Plot')
ax.set_ylabel('Latency (ms)')
ax.set_xlabel('Measurements')
plt.grid(True)
plt.show()
```

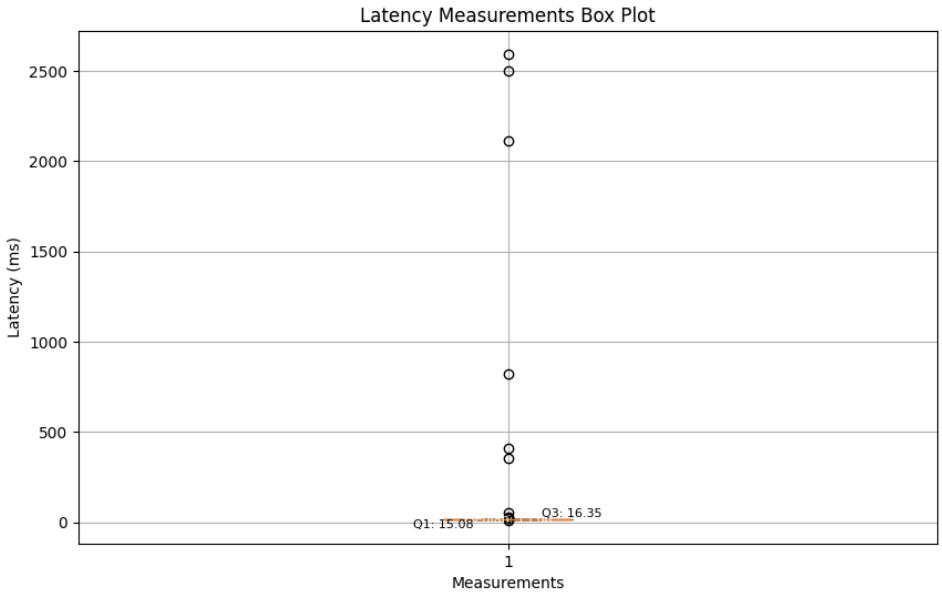
The functionality of this code is to read data from a text file that contains information about query times and connection times, calculate the total latency, and generate a box plot to visualize the latency data. The code starts by extracting information about query times and connection times from the file and then adding them together to calculate the total latency in milliseconds. It proceeds to

sort the total latencies and create a box plot to display the distribution of latency data, with annotations for statistics such as the median and quartiles for better understanding of the data's characteristics. Finally, the generated box plot is displayed using the Matplotlib library.

AS1.d1:

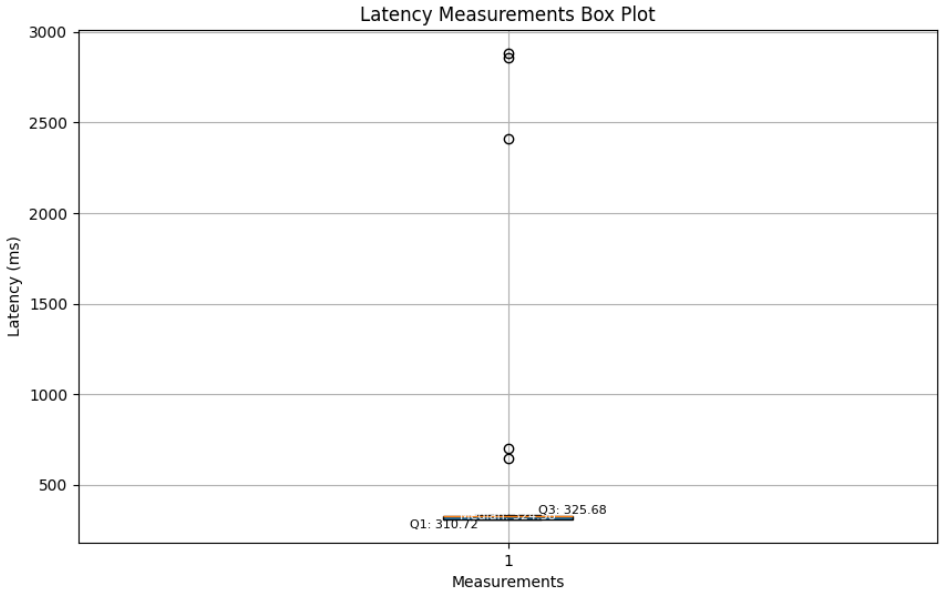


AS1.d2:

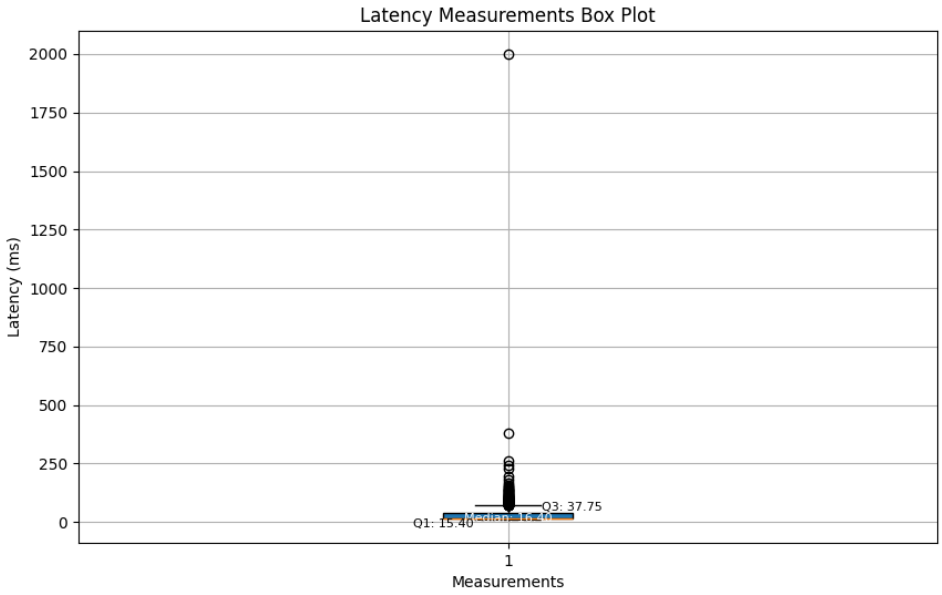


Aalto University
School of Electrical
Engineering

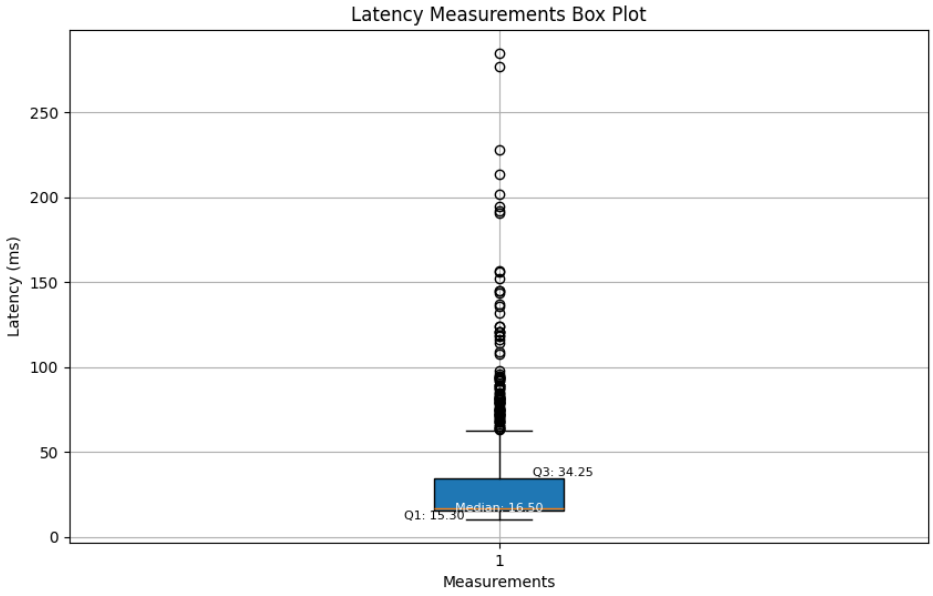
AS1.d3:



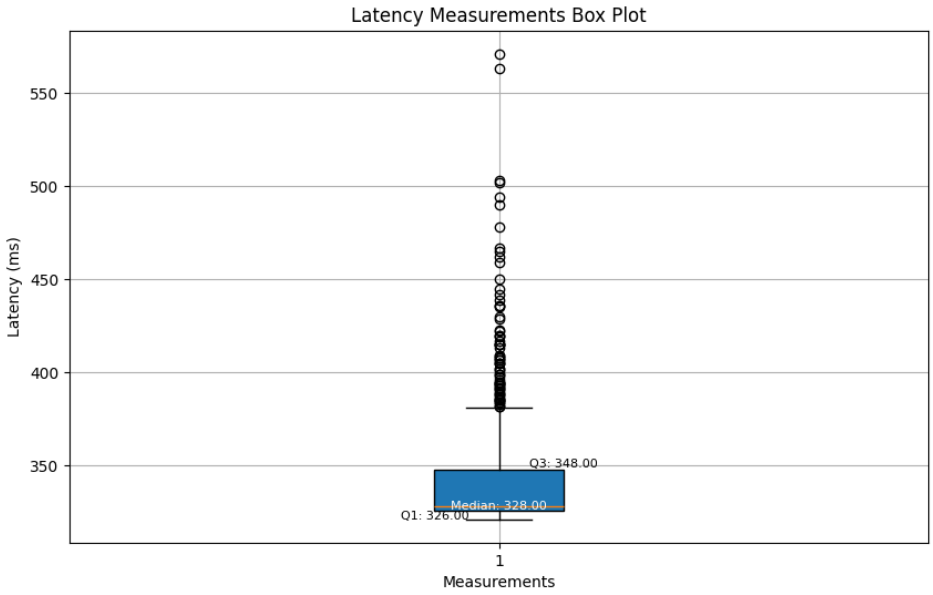
AS1.n1:



AS1.n2:

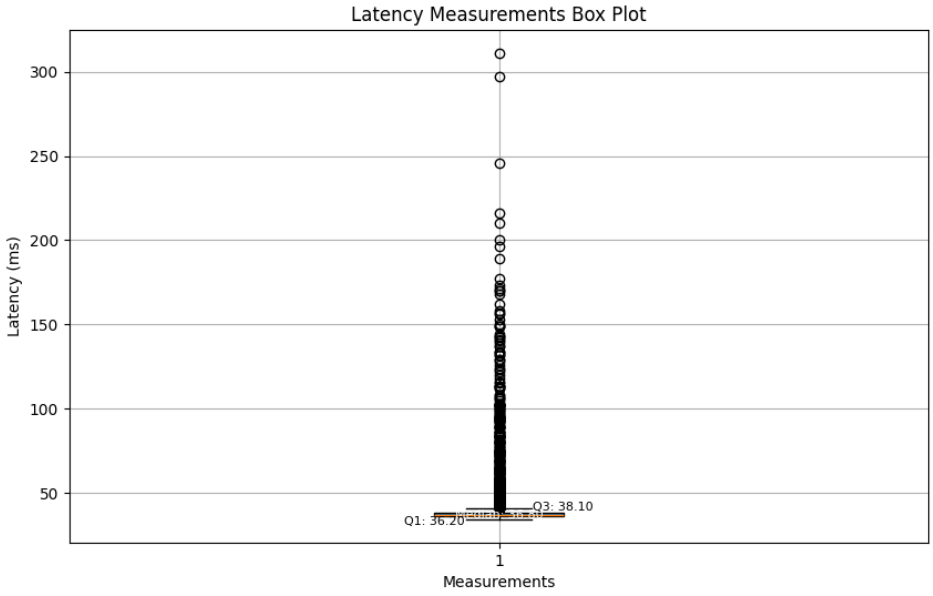

Latency Measurements Box Plot
Latency (ms)
250
200
150
100
50
0
Q3: 34.25
Median: 16.50
Q1: 15.50
1
Measurements

AS1.n3:

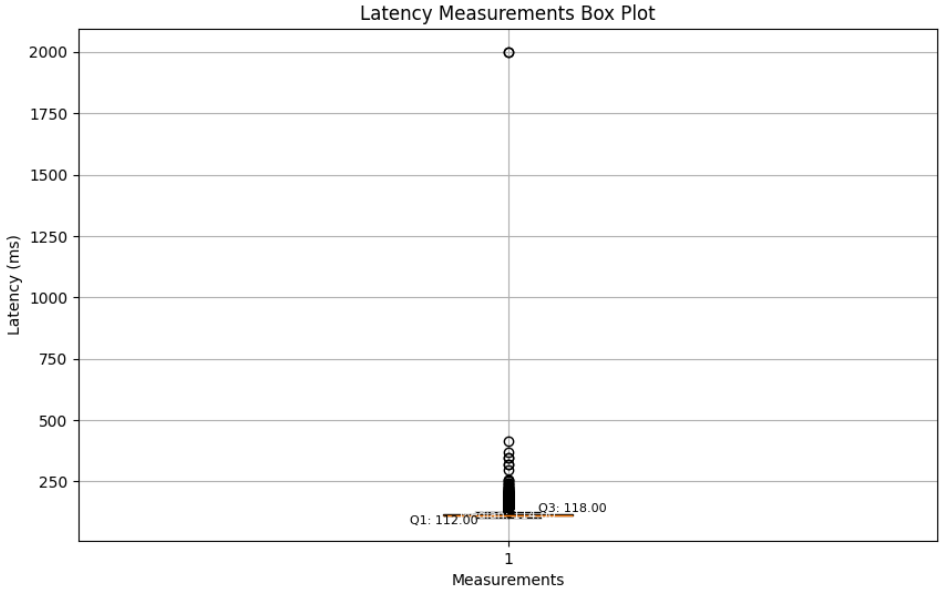

Latency Measurements Box Plot
Latency (ms)
550
500
450
400
350
Q3: 348.00
Median: 328.00
Q1: 326.00
1
Measurements

Aalto University
School of Electrical
Engineering

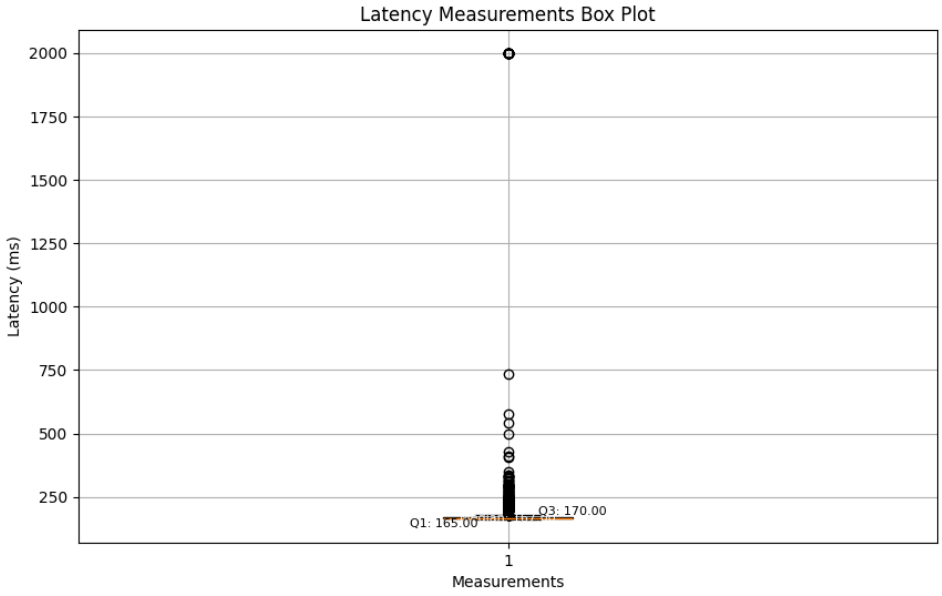
AS1.r1:



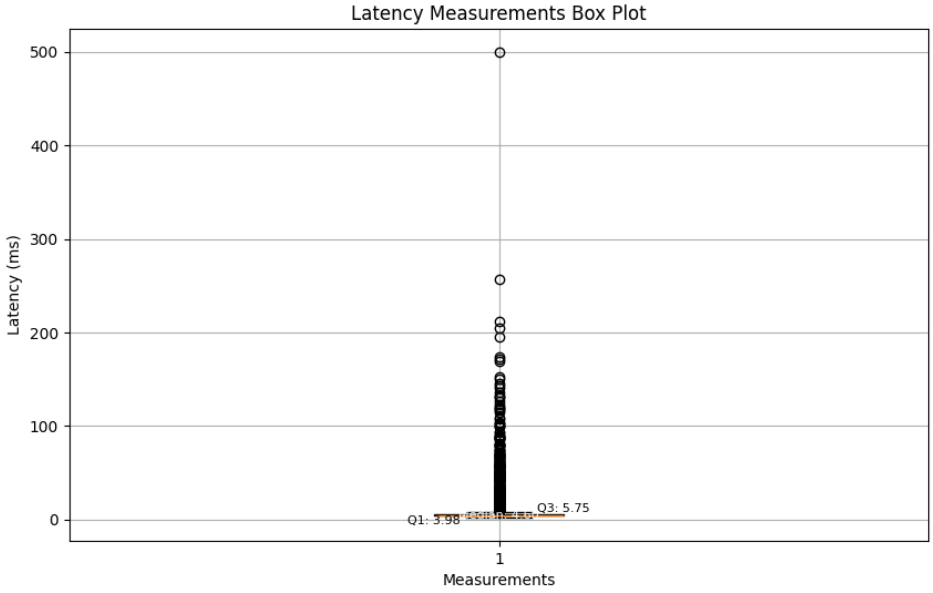
AS1.r2:



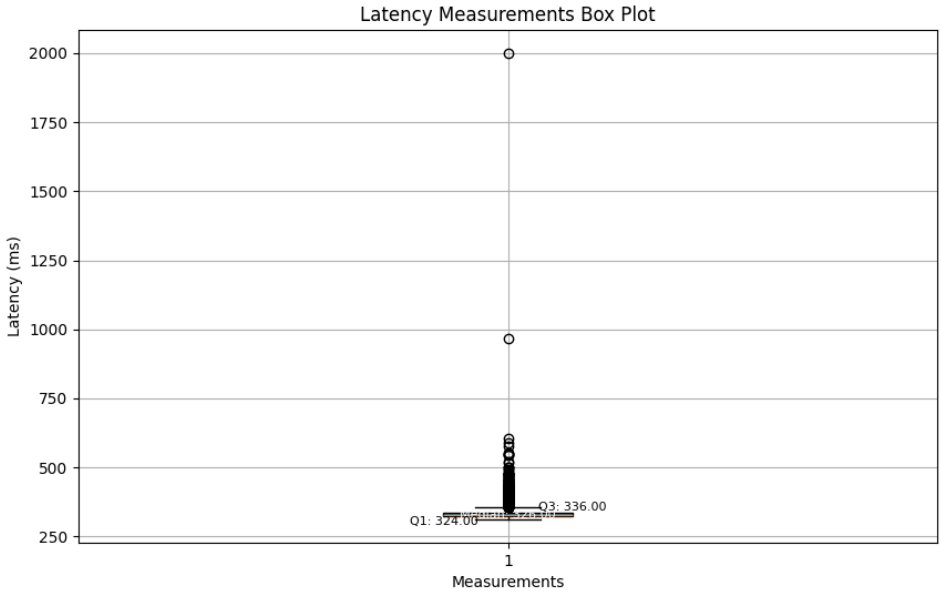
AS1.r3:


Latency Measurements Box Plot
Latency (ms)
2000
1750
1500
1250
1000
750
500
250
Q1: 165.00
Q3: 170.00
1
Measurements

AS1.i1:


Latency Measurements Box Plot
Latency (ms)
500
400
300
200
100
0
Q1: 3.98
Q3: 5.75
1
Measurements

AS1.i2:



- Notable observations

AS1.d1: The median is relatively low, indicating that most latency measurements are concentrated in a lower range. There are several outliers far from the main cluster of data, suggesting occasional peaks in latency.

AS1.d2: The median is also low, but there is a slightly increased interquartile range, indicating more variation in latency. This data set also contains outliers, suggesting occasional higher latencies.

AS1.d3: Both the median and the interquartile range are higher, indicating greater variability in the measurements. The consistent presence of outliers suggests that latency peaks are not uncommon.

AS1.n1: The latency measurements are very tightly grouped, with a very small interquartile range, indicating very consistent latency performance. There are no outliers in this plot, suggesting that latency is stable with no significant spikes.

AS1.n2: The interquartile range is slightly higher than that of AS1.n1, but the latency measurements still cluster tightly. The presence of some outliers indicates that although the server generally performs consistently, there are occasional spikes in latency.

AS1.n3: The median latency is higher and the interquartile range is also wider, indicating less consistency in latency. There are several outliers, indicating that compared to other ICMP measurements, there are more frequent latency spikes.

AS1.r1: The measurements are clustered around a lower median, indicating a generally low latency. The interquartile range (IQR) is quite compact, suggesting consistent latency performance. There are a number of outliers above the upper quartile, indicating occasional spikes in latency.

AS1.r2: The median is higher than in AS1.r1, with a larger IQR, which implies greater variability in latency. The range of the outliers is extensive, suggesting that there are significant deviations in latency at times.

AS1.r3: This plot shows an even higher median and a wide IQR, reflecting even more variability in latency compared to AS1.r2. There is a similar pattern of outliers, which are spread out, showing the potential for large latency peaks.

AS1.i1: The median latency is low, with a very narrow IQR, indicating a very stable and consistent latency profile. While there are outliers, they are not as pronounced or as far from the IQR as in the research server plots.

AS1.i2: The median is slightly higher than that of AS1.i1, with a comparable IQR, suggesting a good consistency in latency, similar to AS1.i1. The presence of outliers is minimal, which shows that the server maintains a stable latency with occasional variations.

- Differences in AS1.d_N and AS1.n_N

The plots for AS1.d1, d2, and d3 exhibit wider interquartile ranges and more outliers, suggesting greater variability in the latency of DNS queries. In contrast, the interquartile ranges for AS1.n1, n2, and n3 are narrower, particularly for AS1.n1 and AS1.n2, indicating more consistent latency for ICMP requests.

The median positions for the DNS servers are higher, especially for AS1.d3, indicating that the average latency is higher compared to ICMP requests. The medians for ICMP are generally lower, suggesting these servers typically respond faster.

The DNS datasets have a greater number of outliers with relatively high latency values, indicating occasional significant delays. There are fewer outliers in the ICMP datasets, and these are closer to the median relative to the interquartile range, suggesting smaller fluctuations in latency for these datasets.

Overall, the latency measurements for DNS queries demonstrate greater variability and instability, while the latency for ICMP requests is relatively lower and more stable. This might reflect the performance characteristics of different types of servers in processing requests.

- PDF and CDF plots

```
import re
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gaussian_kde

# Initialize lists with more descriptive names
query_time_lines = []
curl_lines = []
query_times_ms = []
connect_times_ms = []
delay_ms = []

# Read the file and process each line
with open('files/dig/dns-st.bahnhof.net.reloaded.txt', 'r') as file:
    skip_flag = False
    for line in file:
        striped_line = line.strip()
        if skip_flag:
            skip_flag = False
            continue
        if re.search("Return", striped_line):
            skip_flag = True
            continue
        if re.search("Query", striped_line):
            query_time_lines.append(striped_line)
        if re.search("CURL", striped_line):
            curl_lines.append(striped_line)
```

```
# Extract query times and connect times
for item in query_time_lines:
    item_split = item.split(" ")
    query_time = float(item_split[3])
    query_times_ms.append(query_time)

for item in curl_lines:
    item_split = item.split(" ")
    connect_time = float(item_split[3])
    connect_times_ms.append(connect_time)

# Calculate delays
for i in range(len(query_time_lines)):
    delay = query_times_ms[i] / 1000 + connect_times_ms[i]
    delay_ms.append(delay)

# Calculate probability density function (PDF)
density = gaussian_kde(delay_ms)
xs = np.linspace(min(delay_ms), max(delay_ms), 200)
density.covariance_factor = lambda: .25
density._compute_covariance()

# Plot PDF
plt.figure(figsize=(10, 6))
plt.plot(xs, density(xs))
plt.title('Probability Density Function of Latency')
plt.xlabel('Latency (ms)')
plt.ylabel('Density')
plt.grid(True)
plt.show()

# Calculate cumulative distribution function (CDF)
sorted_data = np.sort(delay_ms)
yvals = np.arange(len(sorted_data)) / float(len(sorted_data) - 1)

# Plot CDF
plt.figure(figsize=(10, 6))
plt.plot(sorted_data, yvals)
plt.title('Cumulative Distribution Function of Latency')
```

```
plt.xlabel('Latency (ms)')
plt.ylabel('Cumulative Probability')
plt.grid(True)
plt.show()
```

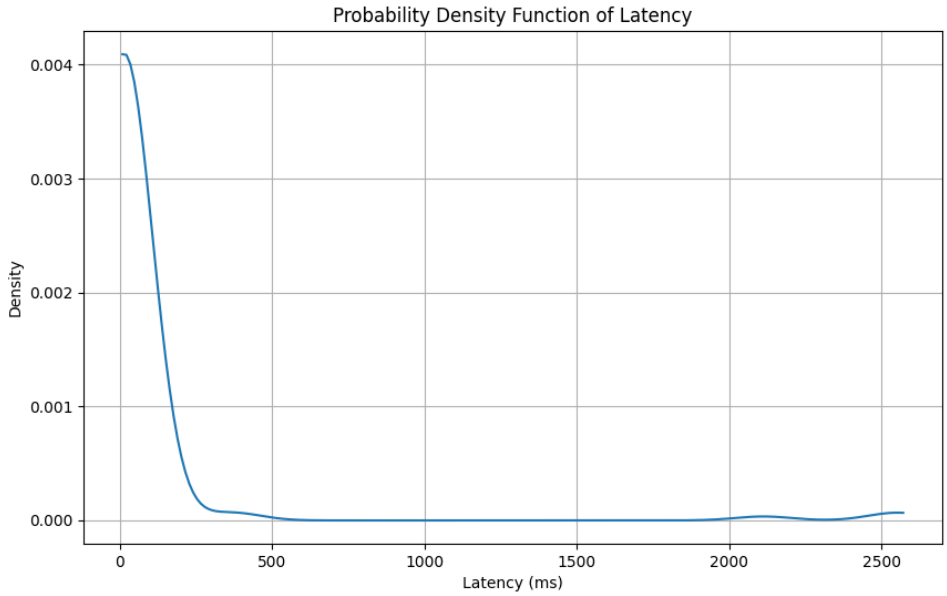
This code reads data from a text file containing latency measurements and calculates both the Probability Density Function (PDF) and Cumulative Distribution Function (CDF) of the latency values. It first processes the input file to extract latency measurements, combines query and connect times to calculate latency delays, and then uses Gaussian kernel density estimation to compute the PDF of the latency values. Finally, it calculates and plots the CDF of the latency values.

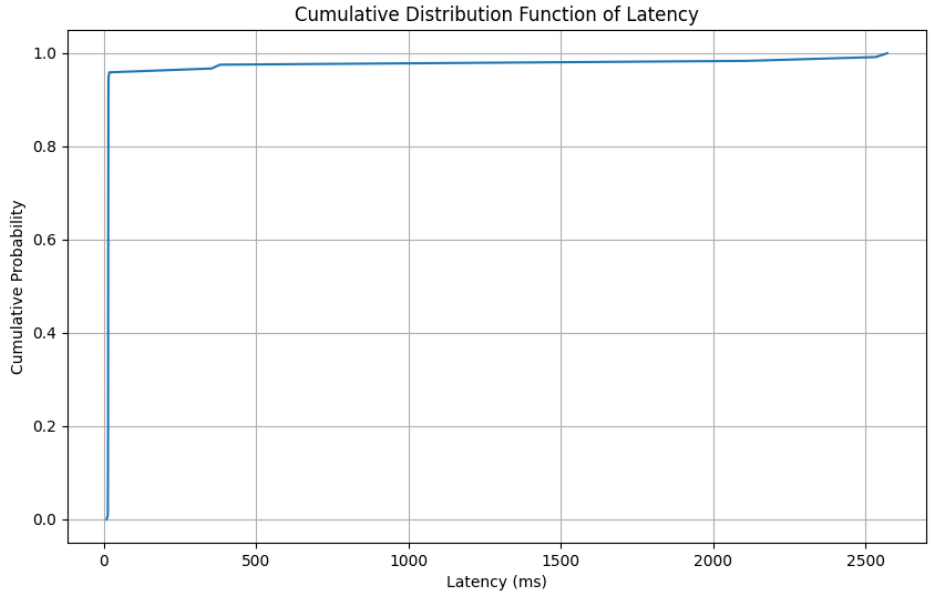
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import gaussian_kde

# Initialize a list to store extracted latency values
latency_values = []

# Open the file and read it line by line
with open('files/ping/southeast-2.dns-au.st.txt', 'r') as file:
    for line in file:
        if 'time=' in line:
            # Find lines containing 'time=' and split the string
            parts = line.split()
            for part in parts:
                if part.startswith('time='):
                    # Extract latency values and remove 'ms'
                    latency = part.split('=')[1].rstrip(' ms')
                    try:
                        # Convert extracted latency values to floating-
point and store them
                        latency_values.append(float(latency))
                    except ValueError:
                        # Skip the value if conversion fails
                        continue

# Calculate the probability density function (PDF)
density = gaussian_kde(latency_values)
xs = np.linspace(min(latency_values), max(latency_values), 200)
```

```
density.covariance_factor = lambda: .25
density._compute_covariance()

# Plot the PDF
plt.figure(figsize=(10, 6))
plt.plot(xs, density(xs))
plt.title('Probability Density Function of Latency')
plt.xlabel('Latency (ms)')
plt.ylabel('Density')
plt.grid(True)
plt.show()

# Calculate the cumulative distribution function (CDF)
sorted_data = np.sort(latency_values)
yvals = np.arange(len(sorted_data)) / float(len(sorted_data) - 1)

# Plot the CDF
plt.figure(figsize=(10, 6))
plt.plot(sorted_data, yvals)
plt.title('Cumulative Distribution Function of Latency')
plt.xlabel('Latency (ms)')
plt.ylabel('Cumulative Probability')
plt.grid(True)
plt.show()
```

This code reads latency measurements from a text file, extracts and processes the latency values, and then calculates and visualizes two statistical functions: the Probability Density Function (PDF) and Cumulative Distribution Function (CDF) of the latency data. It does this by parsing the file to extract latency values, computing a PDF using kernel density estimation, and plotting it to show the probability distribution of latency values. Additionally, it computes a CDF to illustrate the cumulative probability of latency values and displays it using Matplotlib.

AS1.d1:

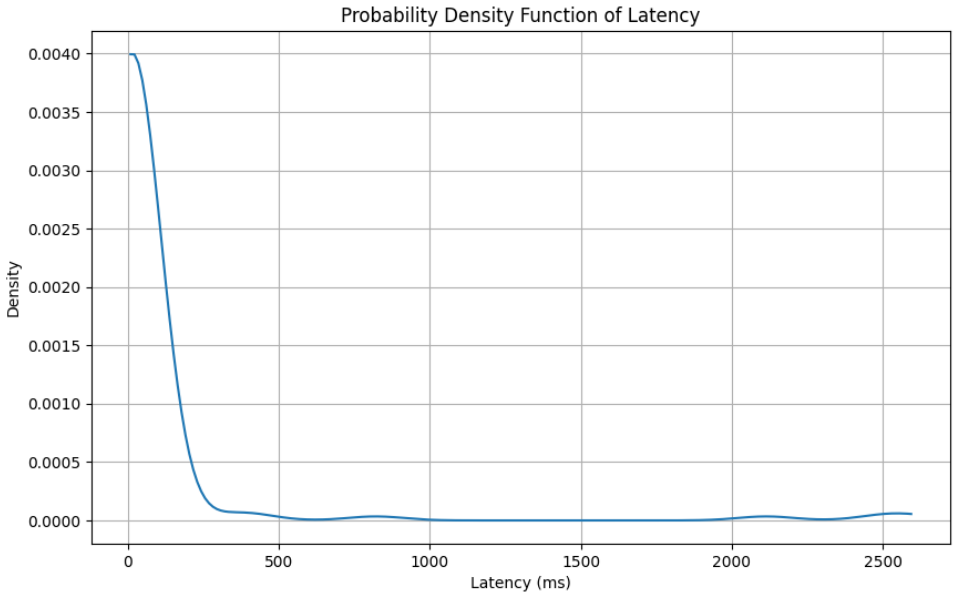


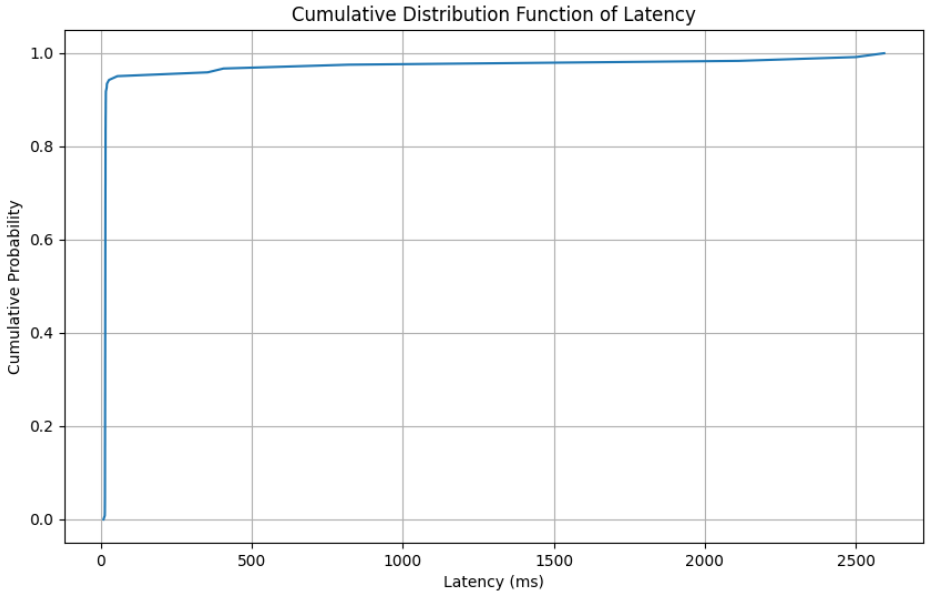


The PDF plot reveals that most latency values are concentrated in the lower latency range, with a peak occurring near zero and then rapidly decreasing, indicating that latency values are primarily concentrated in a small range. The CDF plot shows that almost all latency values are concentrated below 500 milliseconds, with nearly all of the probability mass concentrated in this region,

demonstrating the tightness of the latency distribution.

AS1.d2:

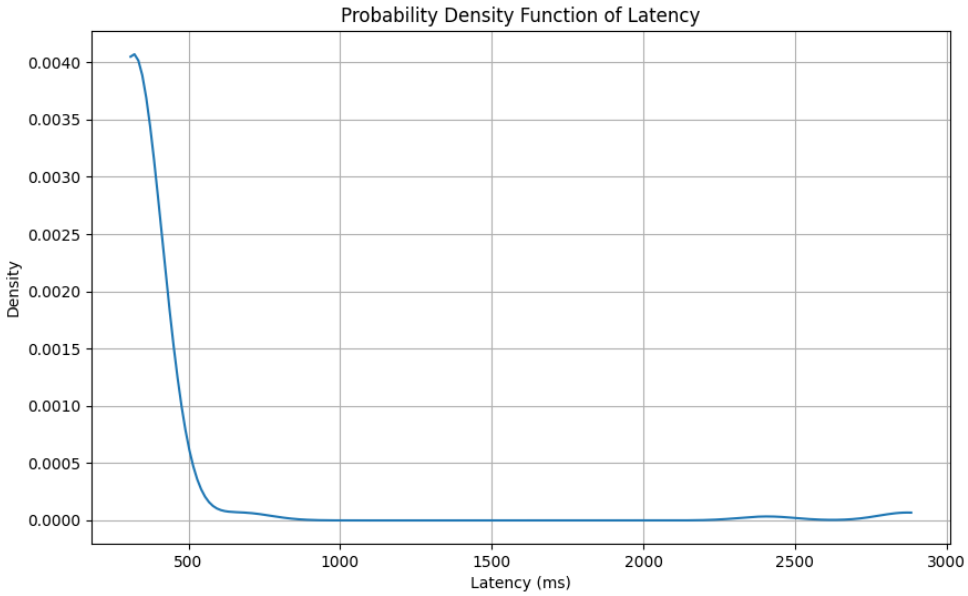


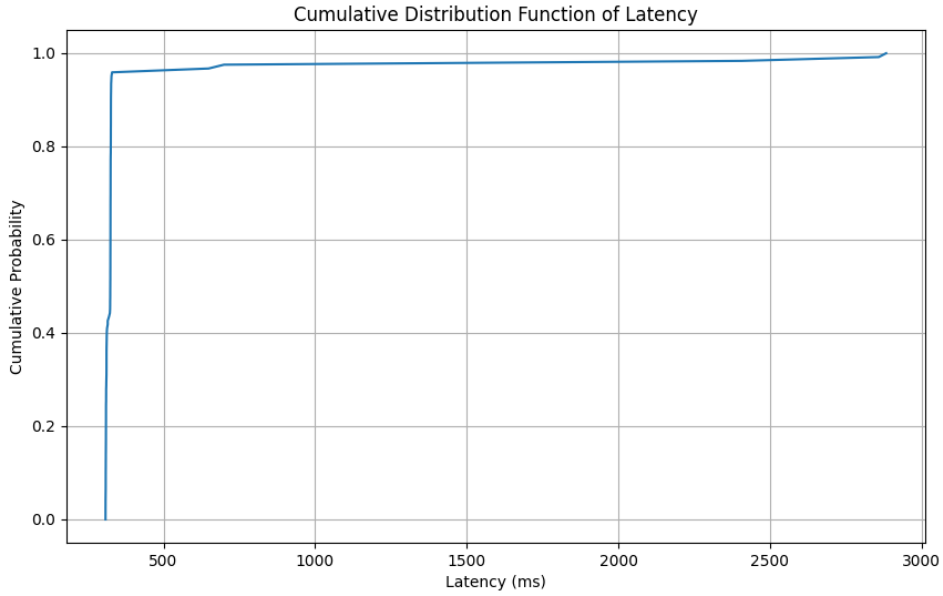


The shape of the PDF is similar to that of AS1.d1, but the tail of the latency value distribution is longer, indicating a wider range of latency values. The shape of the CDF shows that the cumulative probability rises slightly more slowly, indicating that some latency values are distributed in a broader range, but the vast majority of latency values are still concentrated in the lower latency

range.

AS1.d3:

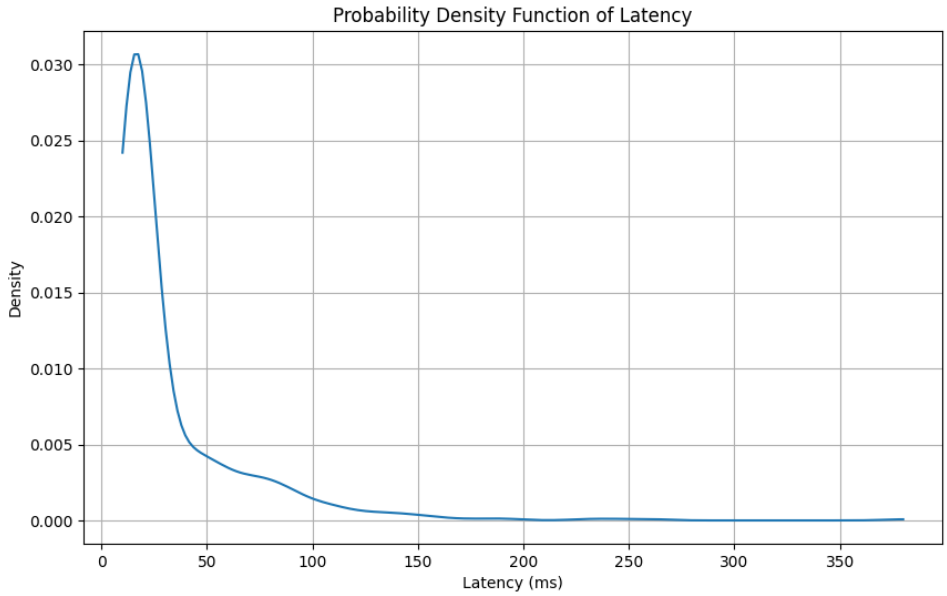


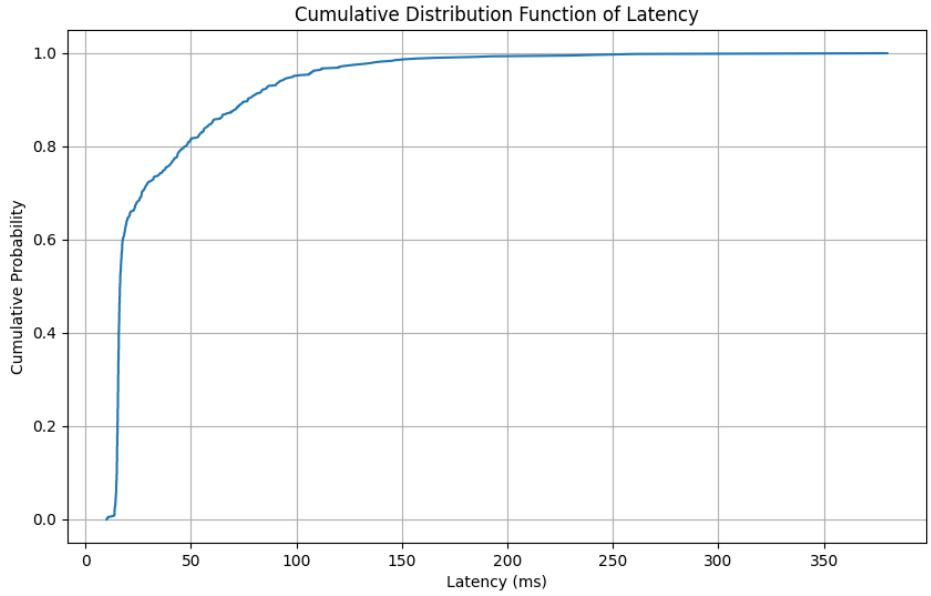


The shape of the PDF is similar to the first two datasets, but the concentration of latency values is not as pronounced as in AS1.d1, and the tail is longer than AS1.d2, indicating greater variability in latency values. The CDF shows a wider range of latency values, with a more uniform distribution of latency values. The cumulative probability rises more gradually, indicating that latency values

are dispersed over a larger range.

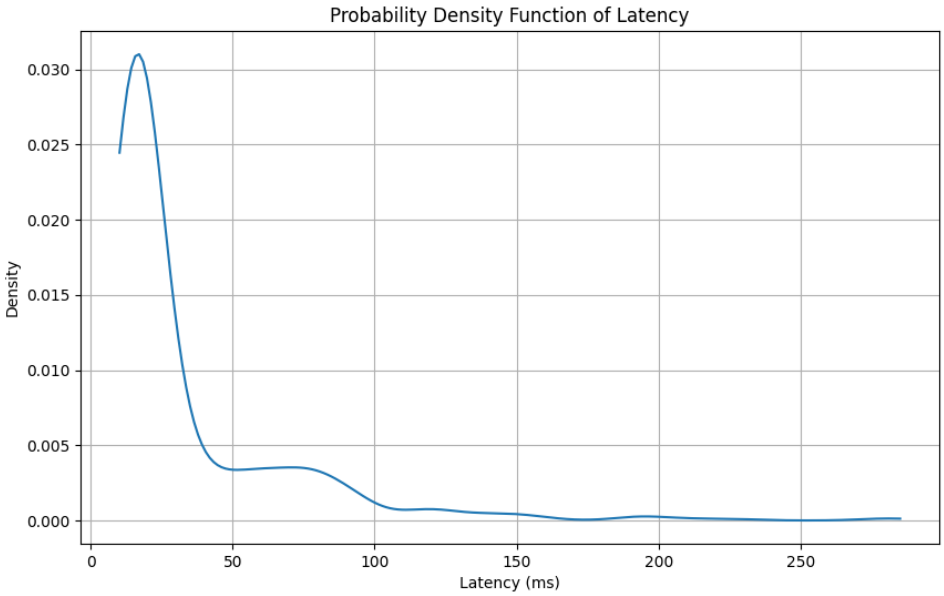
AS1.n1:

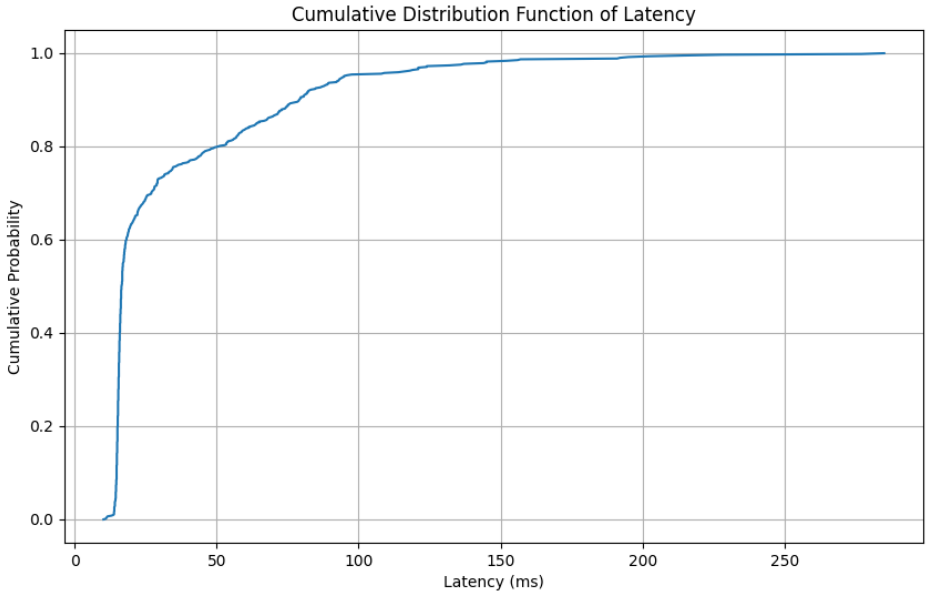




The PDF shows that latency is primarily concentrated in a very low range with a very high peak, indicating that most latency values are very low. The CDF shows rapid accumulation of latency values, with almost all values below 50 milliseconds, indicating that nearly all network requests have very fast response times.

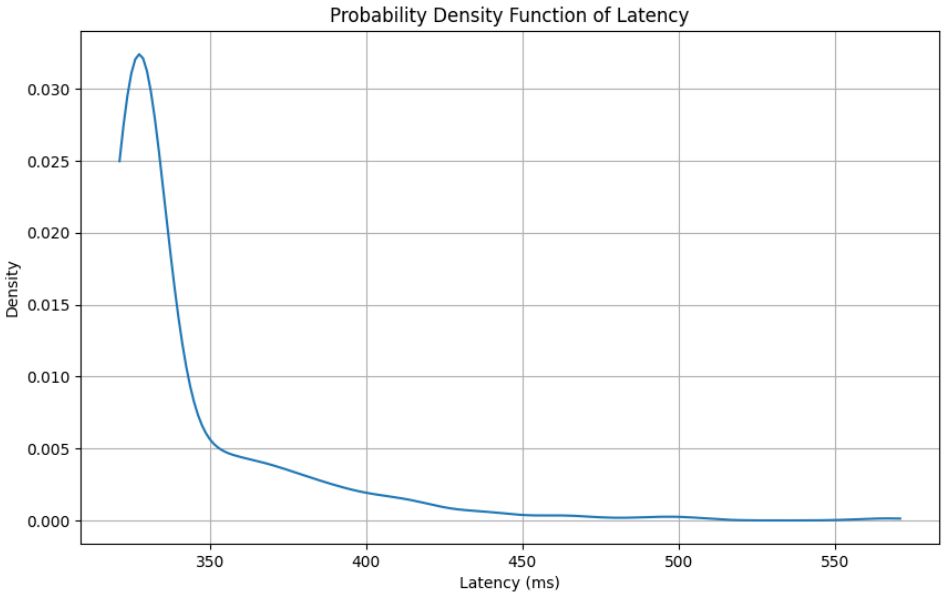
AS1.n2:

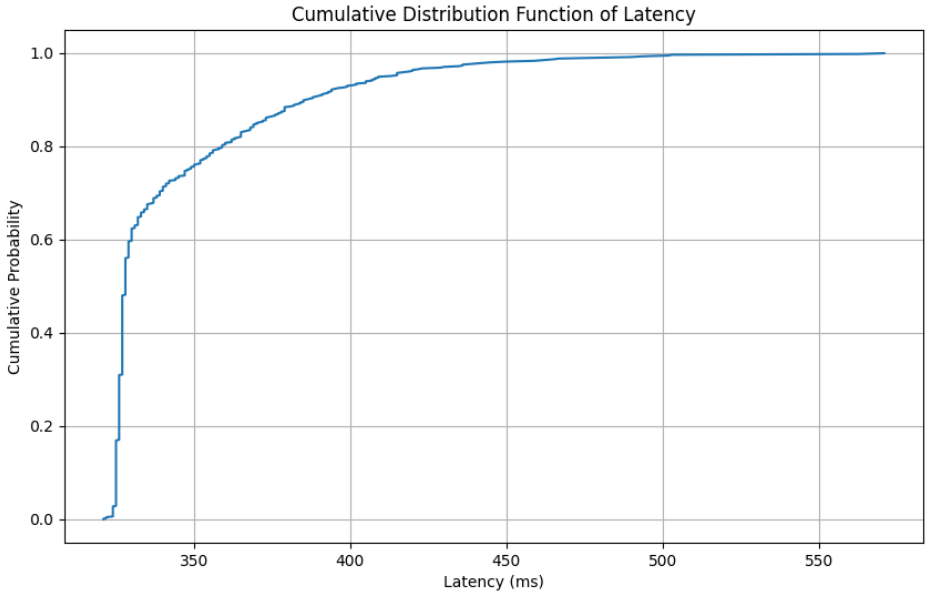




The shape of the PDF is similar to AS1.n1, but with a slightly lower peak, which may suggest a slightly wider distribution of latency values. The shape of the CDF is also similar but rises to high probabilities slightly more slowly, indicating a slightly wider range of latency distribution.

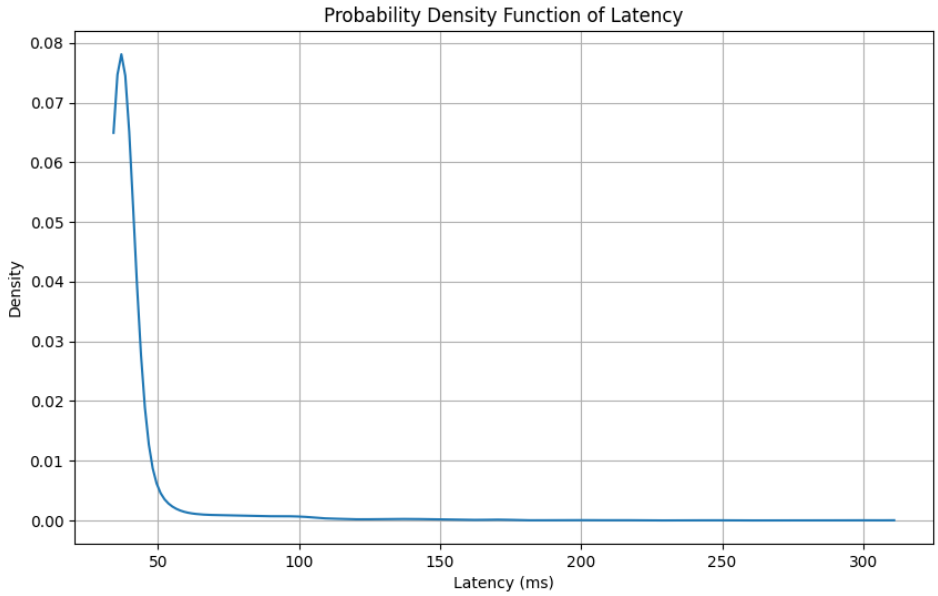
AS1.n3:

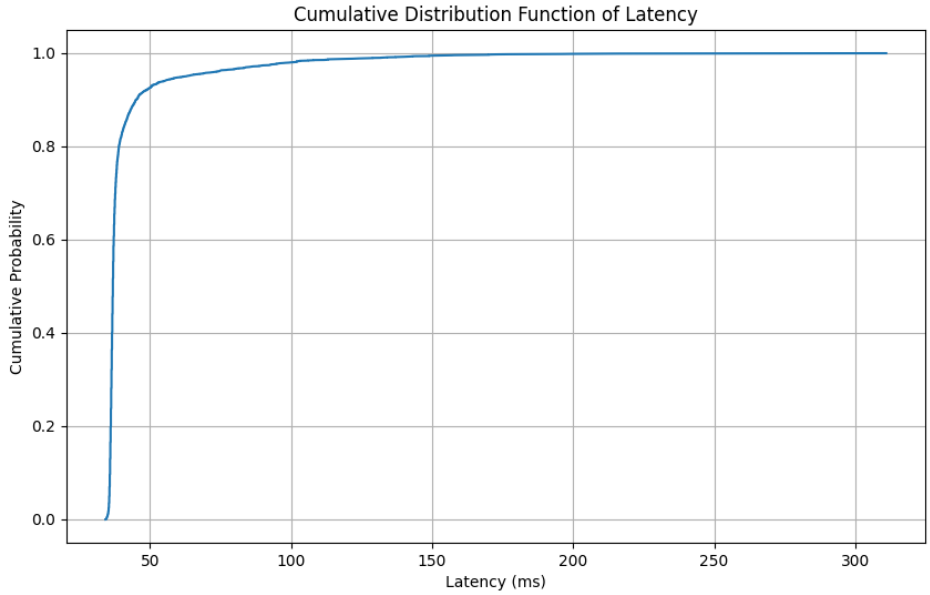




The PDF shows a distribution of latency values similar to the first two datasets but shifted to the right, indicating higher latency values overall. The CDF curve rises more gradually, suggesting that compared to AS1.n1 and AS1.n2, AS1.n3 has more dispersed latency values.

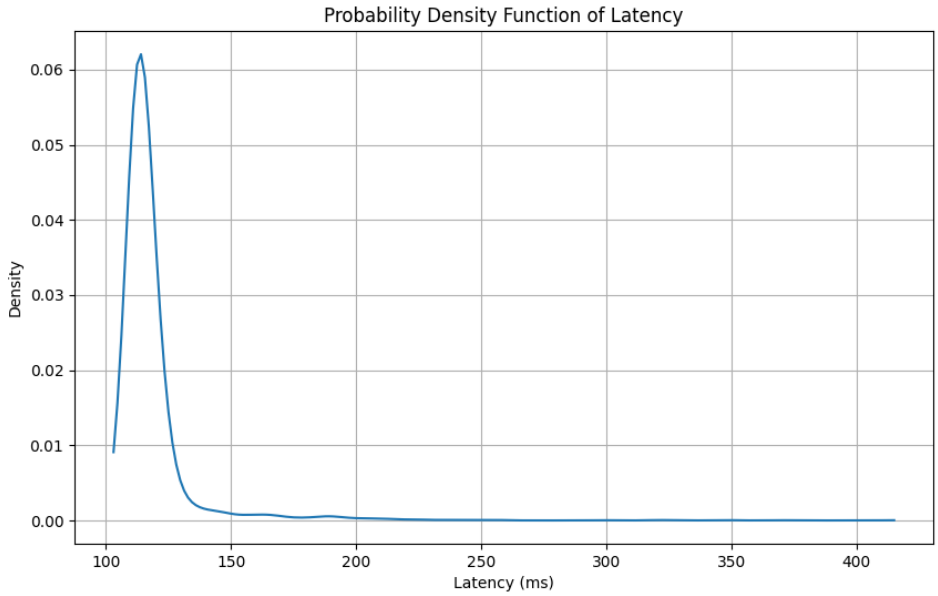
AS1.r1:

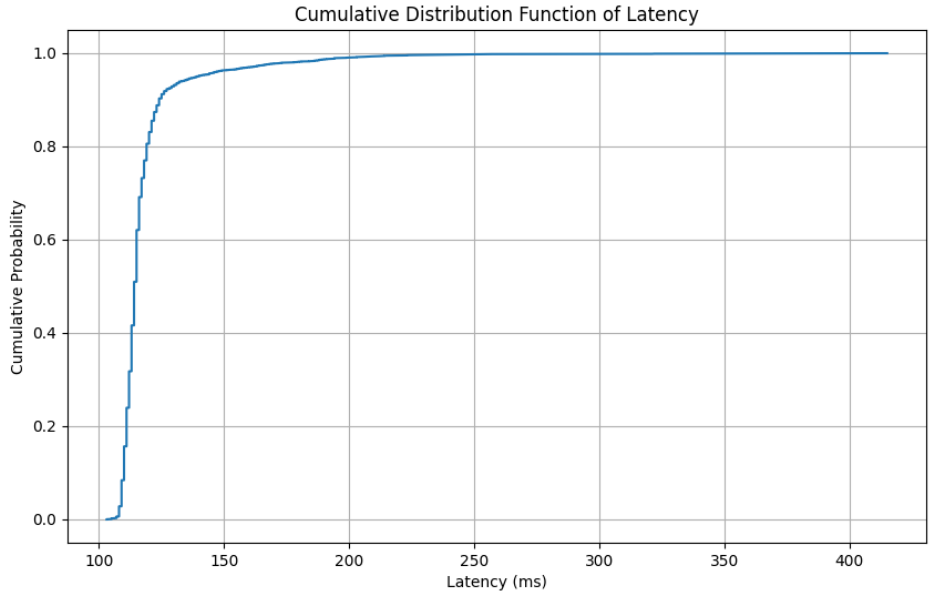




The PDF graph shows that latency values are primarily concentrated in a very low range with a sharp and high peak, indicating consistent and low latency values. The CDF graph indicates that almost all latency values are concentrated below 100 milliseconds, suggesting that the response times for nearly all requests are very short.

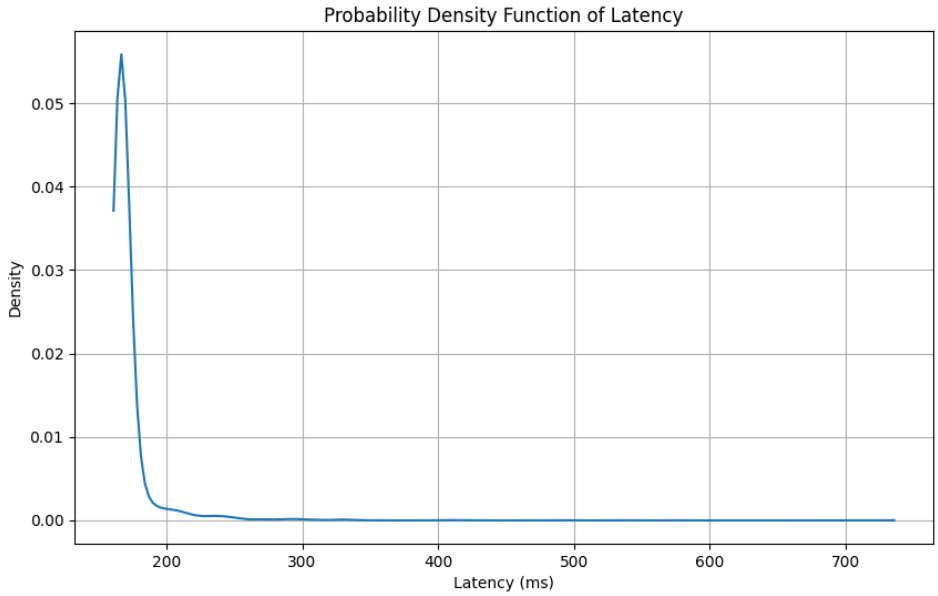
AS1.r2:

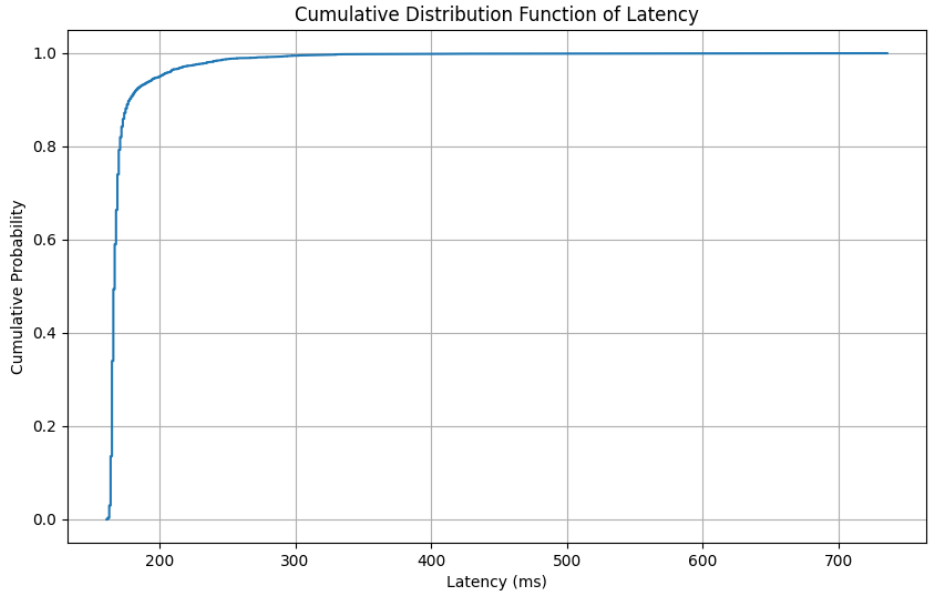




The PDF graph shows a slightly broader distribution of latency values, although there is still a peak, it's not as sharp as AS1.r1. The curve of the CDF graph doesn't rise as steeply as AS1.r1, suggesting that the latency values are distributed over a wider range but still mostly below 100 milliseconds.

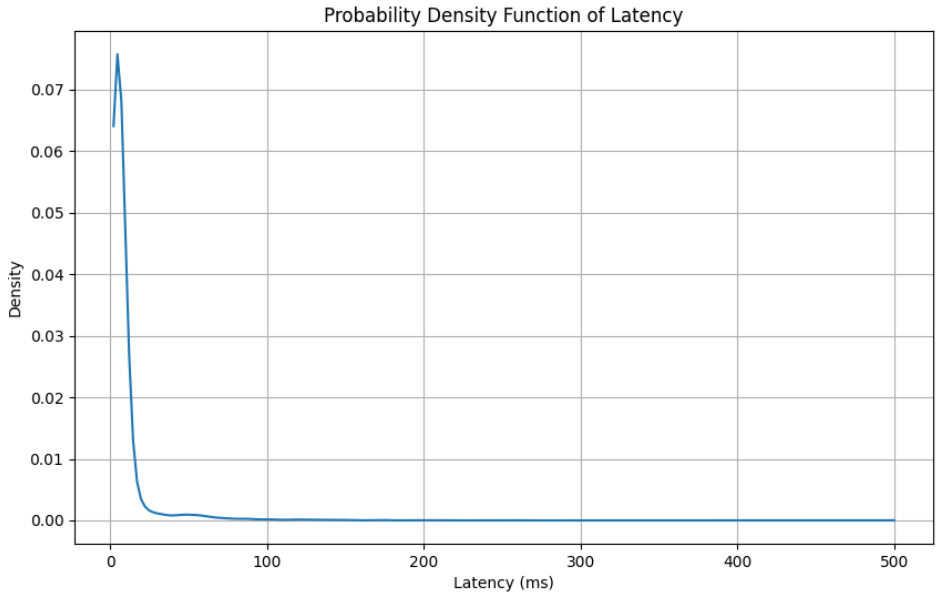
AS1.r3:

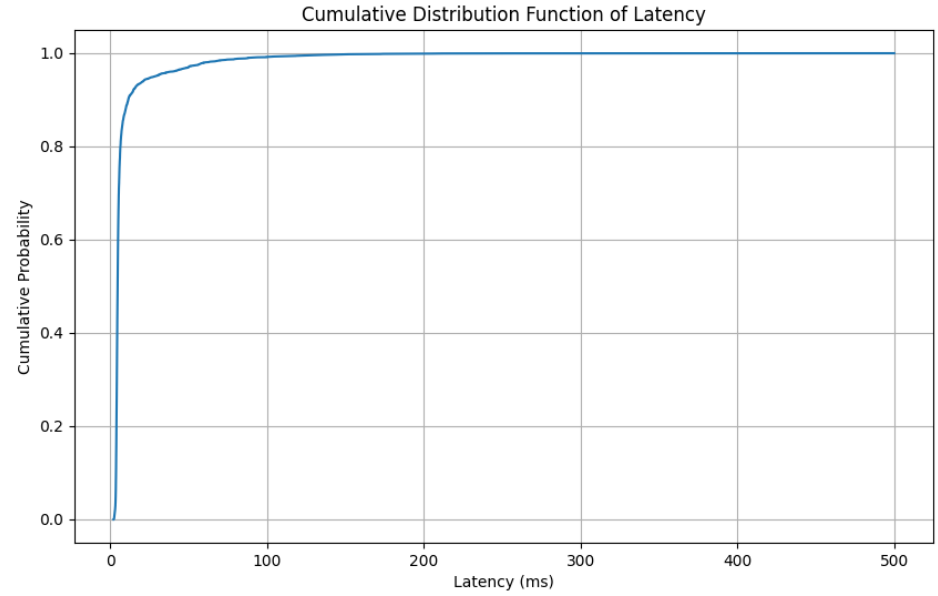




The PDF graph shows a slightly broader distribution of latency values, although there is still a peak, it's not as sharp as AS1.r1. The curve of the CDF graph doesn't rise as steeply as AS1.r1, suggesting that the latency values are distributed over a wider range but still mostly below 100 milliseconds.

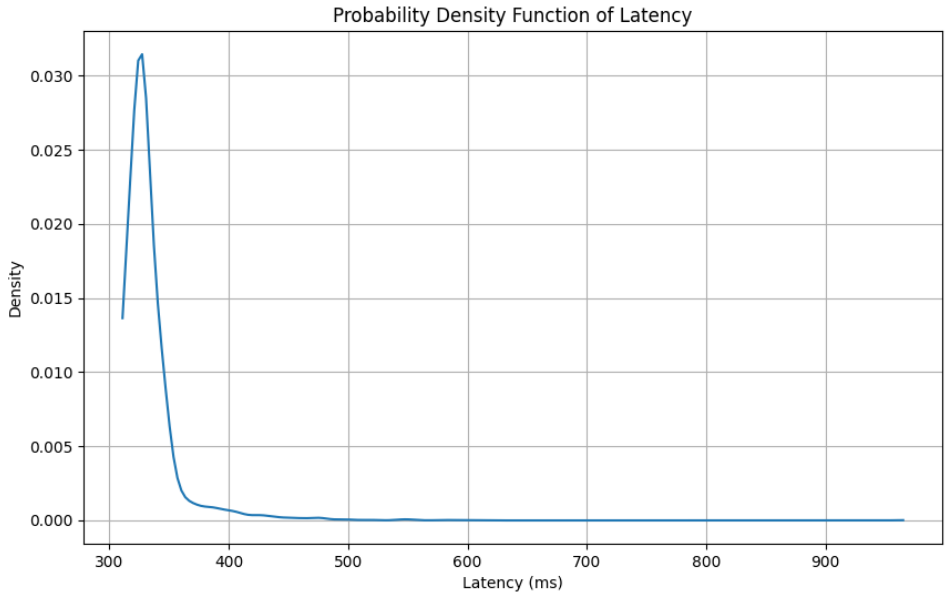
AS1.i1:

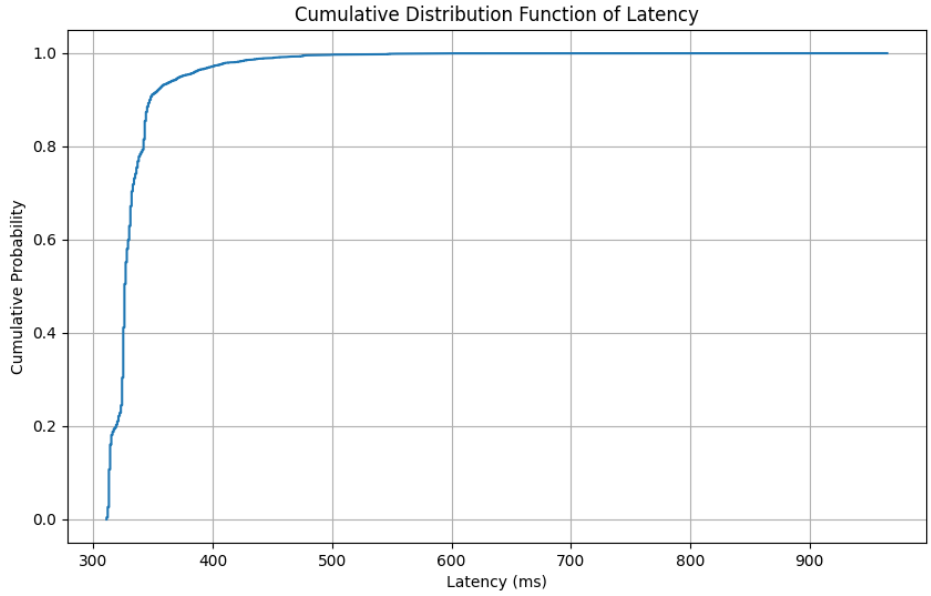




The PDF graph displays an extremely sharp peak, almost at 0 milliseconds, indicating that the majority of latencies are very low, with almost no delay. The CDF graph rapidly rises to values close to 1, further confirming that most latencies are very low, and the cumulative distribution of latencies quickly approaches 100%.

AS1.i2:





The PDF graph shows a wider distribution of latencies compared to AS1.i1, with a lower peak, indicating that latency values are relatively more dispersed, with fewer extremely low latency measurements. The CDF graph doesn't rise as rapidly as AS1.i1, suggesting that the cumulative distribution of latencies increases over a broader range, indicating greater dispersion of latency

values.

- A table summarizing delay distributions for all AS1.x data sets

Data set	First packet delay (ms)	Mean delay (ms)	Proportion of packets outside 1000 ms (%)	Distance between 0.95 and 0.5 (ms)
AS1.d1	13.77	79.11	2.44	1.50
AS1.d2	14.28	86.79	2.44	37.19
AS1.d3	323.28	383.27	2.44	3.93
AS1.n1	15.60	33.02	0.00	82.07
AS1.n2	31.60	33.51	0.00	78.33
AS1.n3	339.00	343.63	0.00	82.20
AS1.r1	36.00	41.15	0.00	25.03
AS1.r2	119.00	118.17	0.00	25.00
AS1.r3	165.00	171.94	0.00	33.00
AS1.i1	5.15	8.65	0.00	23.52
AS1.i2	348.00	333.01	0.00	48.00

```
import re
import numpy as np

# Initialize a list to store latency differences
diff = []

# Open the file and read it line by line
with open('files/ping/blr1.iperf.comnet-student.eu.txt', 'r') as
file:
    for line in file:
        if 'time=' in line:
            # Find lines containing 'time=' and split the string
            parts = line.split()
            for part in parts:
                if part.startswith('time='):
                    # Extract latency differences and remove 'ms'
                    latency_diff = part.split('=')[1].rstrip(' ms')
```

```
                try:
                    # Convert extracted latency differences to
floating-point and store them
                    diff.append(float(latency_diff))
                except ValueError:
                    # Skip the value if conversion fails
                    continue

# Calculate First Packet Delay
first_packet_delay = diff[0]

# Calculate Mean Delay
mean_delay = np.mean(diff)

# Calculate the proportion of delays exceeding 1000 ms
proportion_outside = np.sum(np.array(diff) > 1000.0) / len(diff)

# Calculate the distance between quantiles
quantile_95 = np.quantile(diff, 0.95)
quantile_50 = np.quantile(diff, 0.5)
distance_between_quantiles = quantile_95 - quantile_50

# Print results
print(f"First Packet Delay: {first_packet_delay} ms")
print(f"Mean Delay: {mean_delay} ms")
print(f"Proportion of Delays Over 1000ms: {proportion_outside *
100:.2f}%")
print(f"Distance Between 0.95 and 0.50 Quantiles:
{distance_between_quantiles} ms")
```

This code reads latency differences from a log file, calculates various statistics on these differences, and prints the results. It specifically extracts and analyzes latency measurements in the form of differences, calculates the first packet delay, mean delay, proportion of delays exceeding 1000 ms, and the distance between the 0.95 and 0.50 quantiles.

```
import re
import numpy as np

# Initialize lists with more descriptive names
```

```
query_time_lines = []
curl_lines = []
query_times_ms = []
connect_times_ms = []
diff = []

# Open and read the file
with open('files/dig/southeast-2.dns-au.st.reloaded.txt', 'r') as
file:
    skip_flag = False
    for line in file:
        striped_line = line.strip()
        if skip_flag:
            skip_flag = False
            continue
        if re.search("Return", striped_line):
            skip_flag = True
            continue
        if re.search("Query", striped_line):
            query_time_lines.append(striped_line)
        if re.search("CURL", striped_line):
            curl_lines.append(striped_line)

# Extract query times and connect times
for item in query_time_lines:
    item_split = item.split(" ")
    query_time = float(item_split[3])
    query_times_ms.append(query_time)

for item in curl_lines:
    item_split = item.split(" ")
    connect_time = float(item_split[3])
    connect_times_ms.append(connect_time)

# Calculate differences (formerly known as delays) in milliseconds
for i in range(len(query_time_lines)):
    difference = query_times_ms[i] / 1000 + connect_times_ms[i]
    diff.append(difference)

# Calculate First Packet Delay
```

```
first_packet_delay = diff[0]

# Calculate Mean Delay
mean_delay = np.mean(diff)

# Calculate the proportion of packets exceeding 1000 ms
proportion_outside = np.sum(np.array(diff) > 1000.0) / len(diff)

# Calculate the distance between quantiles
quantile_95 = np.quantile(diff, 0.95)
quantile_50 = np.quantile(diff, 0.5)
distance_between_quantiles = quantile_95 - quantile_50

# Print results
print(f"First Packet Delay: {first_packet_delay} ms")
print(f"Mean Delay: {mean_delay} ms")
print(f"Proportion of Packets Over 1000ms: {proportion_outside *
100:.2f}%")
print(f"Distance Between 0.95 and 0.50 Quantiles:
{distance_between_quantiles} ms")
```

The purpose of this code is to extract and analyze query and connection time data from a log file, calculate the differences (latencies) between these time data, compute the latency of the first packet, the mean latency, the proportion of latencies exceeding 1000 milliseconds, and the distance between the 0.95th and 0.50th percentiles. Finally, it prints the calculated results.

2. Latency data time series

- Plot time series of each data set AS1.x

```
import re
import numpy as np
import matplotlib.pyplot as plt

query_time_lines = []
curl_lines = []
timestamps = []

with open('files/dig/dns-st.bahnhof.net.txt', 'r') as file:
```

```
    skip_flag = False
    for line in file:
        striped_line = line.strip()
        if skip_flag:
            skip_flag = False
            continue
        if re.search("Return", striped_line):
            skip_flag = True
            timestamps.pop(-1)
            continue
        if re.search("Query", striped_line):
            query_time_lines.append(striped_line)
        if re.search("CURL", striped_line):
            curl_lines.append(striped_line)
        if re.search("2023-", striped_line):
            timestamps.append(striped_line)

query_time_ms = []
connect_time_ms = []

for item in query_time_lines:
    item_split = item.split(" ")
    query_time = float(item_split[3])
    query_time_ms.append(query_time)

for item in curl_lines:
    item_split = item.split(" ")
    connect_time = float(item_split[3])
    connect_time_ms.append(connect_time)

latency_ms = []
for i in range(0, len(query_time_lines)):
    latency = query_time_ms[i] / 1000 + connect_time_ms[i]
    latency_ms.append(latency)

# Create a time series plot
plt.figure(figsize=(20, 8))  # Make the plot wider
plt.plot(timestamps, latency_ms, linestyle='-')
plt.xlabel('Date and Time')
plt.ylabel('Latency (ms)')
```

```
plt.title('DNS Latency Time Series')
plt.grid(True)

# Format x-axis to display both date and time, every 10th timestamp
plt.xticks(rotation=90, fontsize=8)

# Show the plot
plt.show()
```

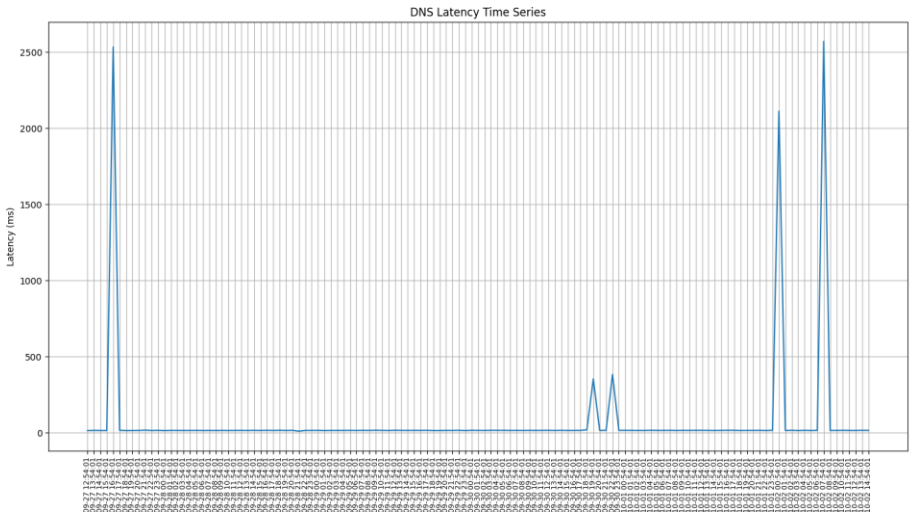
This code reads a log file containing DNS query and CURL request information, extracts relevant data, and calculates the latency for each event. It then creates a time series plot to visualize how DNS latency varies over time.

```
import re
import matplotlib.pyplot as plt
from datetime import datetime

# Initialize lists to store timestamps, TTLs, and time values
timestamps = []
ttls = []
time_values = []

# Open the file for reading
with open('files/ping/bjl-gm.ark.caida.org.txt', 'r') as file:
    # Read each line in the file
    for line in file:
        # Use regular expressions to extract timestamp, TTL, and time
value
        match =
re.search(r'\[(\d+\.\d+)\].*ttl=(\d+).*time=(\d+(\.\d+)?)', line)
        if match:
            timestamp = float(match.group(1))
            ttl = int(match.group(2))
            time = float(match.group(3))
            timestamps.append(timestamp)
            ttls.append(ttl)
            time_values.append(time)

# Convert timestamps to datetime objects with both date and time
```

```
datetime_timestamps =
[datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S') for ts in
timestamps]

# Create a time series plot for latency with transparent background
plt.figure(figsize=(12, 6))
plt.plot(datetime_timestamps, time_values, linestyle='-')
plt.xlabel('Date and Time')
plt.ylabel('Time (ms)')
plt.title('Latency Time Series')
plt.grid(axis="y")
plt.xticks(rotation=45, fontsize=0.1)
plt.show()
```

The purpose of this code is to read network latency data from a text file, where each line contains timestamp, TTL (Time to Live), and time value (latency) information. It uses regular expressions to parse this information and converts timestamps into date and time format. Then, it creates a time series plot that displays the trend of latency changing over time.

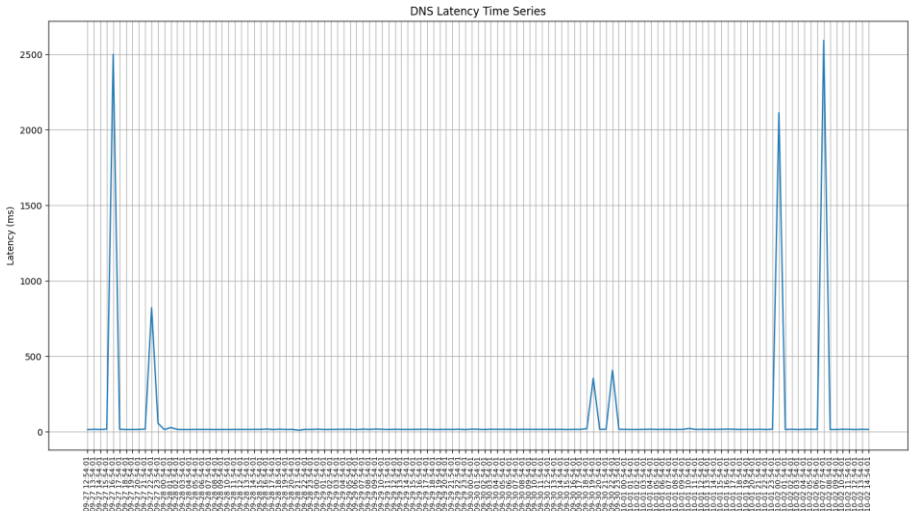
AS1.d1:



This chart displays the DNS query latency time series for Name Server 1. The graph shows that for the most part, latency is very low, close to 0 milliseconds, but there are several abnormal peaks, indicating moments of sharp increases in latency. These peaks could be due to network congestion, the server reaching its processing capacity limit, or network failures, among other reasons. The

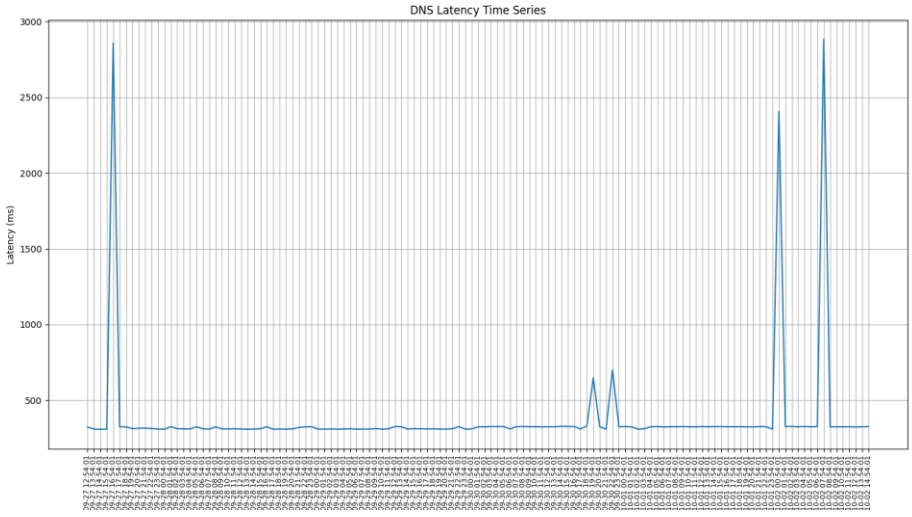
highest peaks in latency are close to 2500 milliseconds.

AS1.d2:



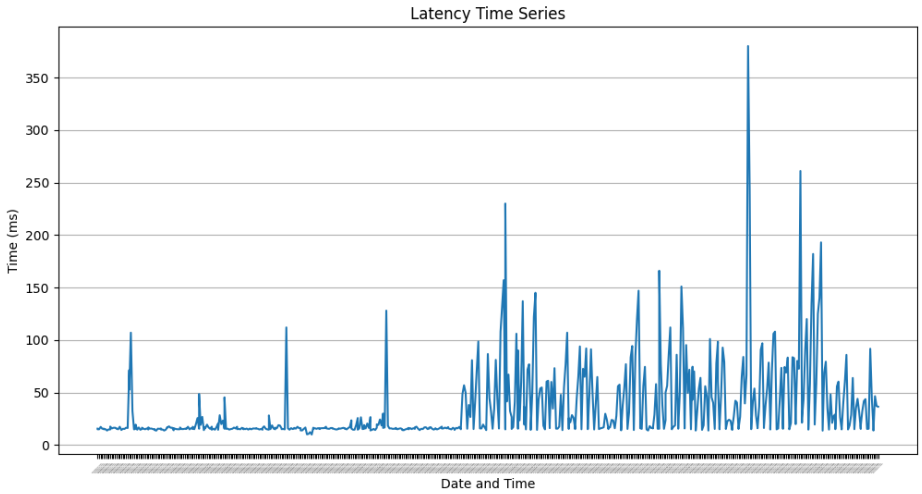
This chart displays the DNS query latency time series for Name Server 1. The graph shows that for the most part, latency is very low, close to 0 milliseconds, but there are several abnormal peaks, indicating moments of sharp increases in latency. These peaks could be due to network congestion, the server reaching its processing capacity limit, or network failures, among other reasons. The highest peaks in latency are close to 2500 milliseconds.

AS1.d3:



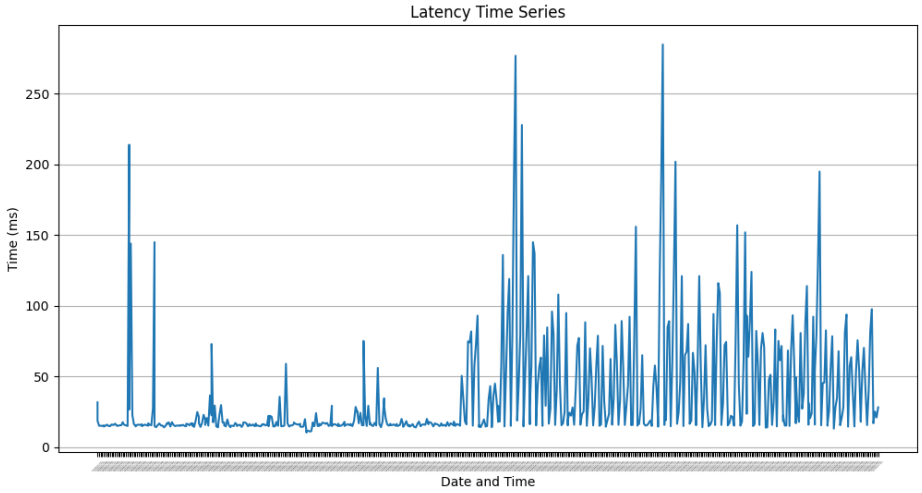
The peaks in the DNS query latency chart for Name Server 3 are more frequent, and the peak heights exceed those of the previous two servers, with the highest even surpassing 2500 milliseconds. Frequent high latency may indicate more serious stability issues with this server, possibly due to the server's location or longer network paths.

AS1.n1:



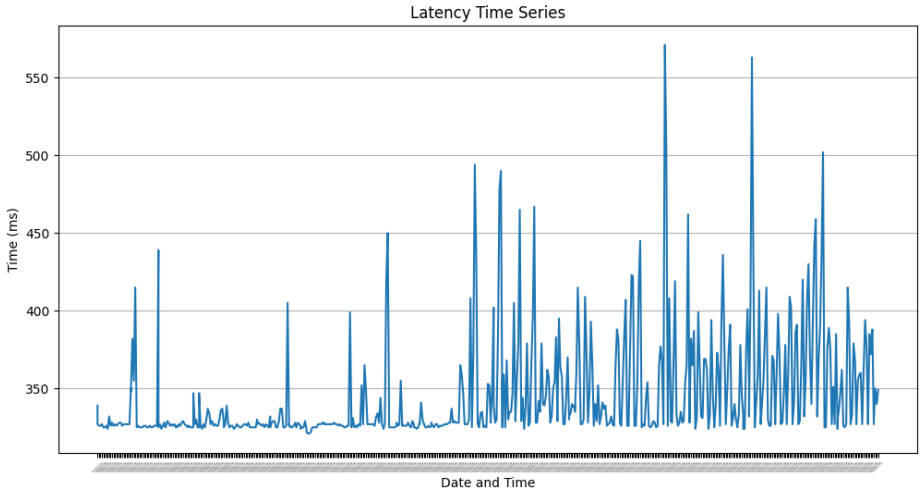
For ICMP queries to Name Server 1, the overall latency is lower, which may imply that the network connection itself is relatively stable. Nevertheless, there are several significant increases in latency, which could be the result of transient network issues or ICMP traffic being treated with a lower priority.

AS1.n2:



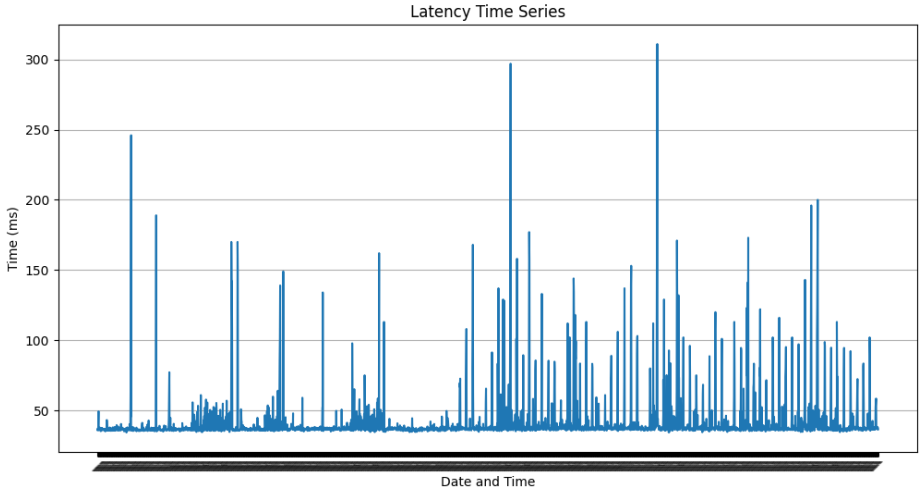
The ICMP latency chart for Name Server 2 shows a lower baseline latency, similar to Server 1. Despite a few sudden increases in latency, the network generally displays good stability. However, any peak in latency could affect applications that require real-time responses.

AS1.n3:



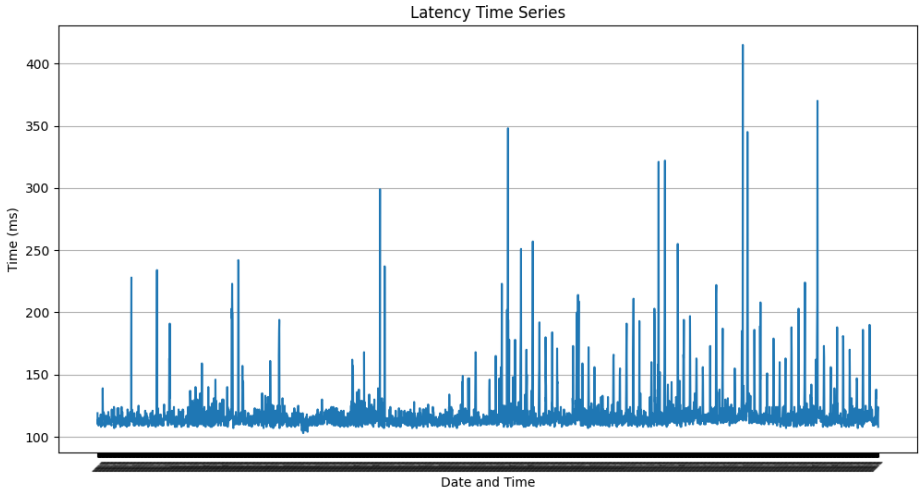
The ICMP latency chart for Name Server 3 shows more frequent high peaks, similar to its DNS query latency graph, suggesting that this server might have more stability issues. The peak latency often exceeds 500 milliseconds, which is unacceptable for any network service that requires a quick response.

AS1.r1:



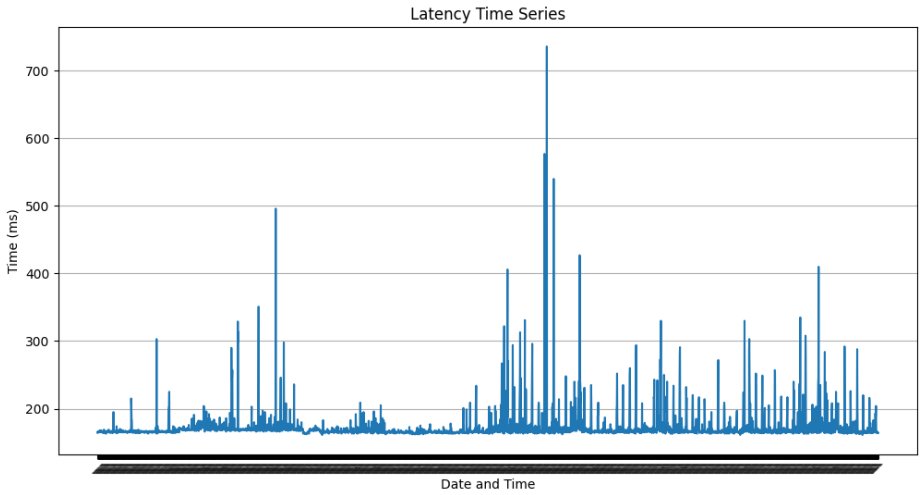
The latency baseline is low, most of the time maintaining around 50 milliseconds, but there are several spikes indicating brief increases in latency, with the highest peaks nearing 300 milliseconds. These peaks could be caused by temporary network fluctuations, instantaneous server load increases, or short-term issues with the network infrastructure.

AS1.r2:



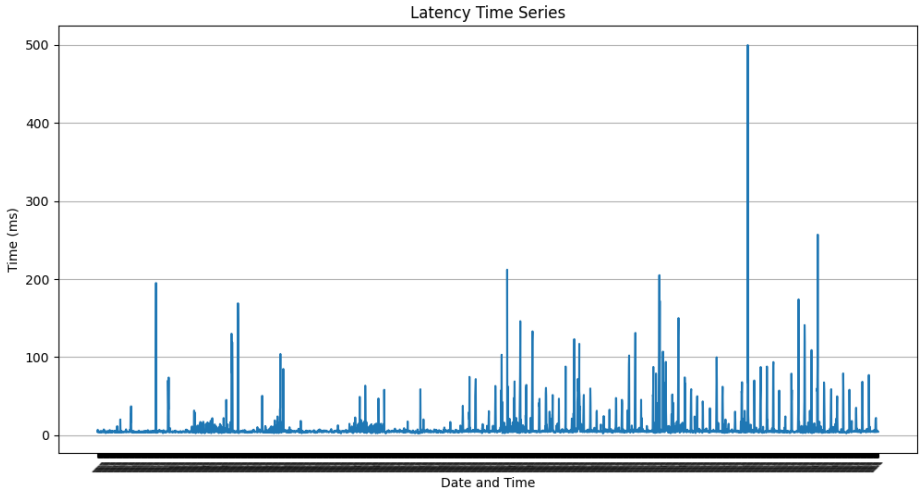
The chart for the second research server shows more frequent peaks, and these peaks are typically higher, reaching above 400 milliseconds. This suggests that the server may face more network stability challenges than the first server. The frequency of high peaks could indicate more common network connection issues.

AS1.r3:



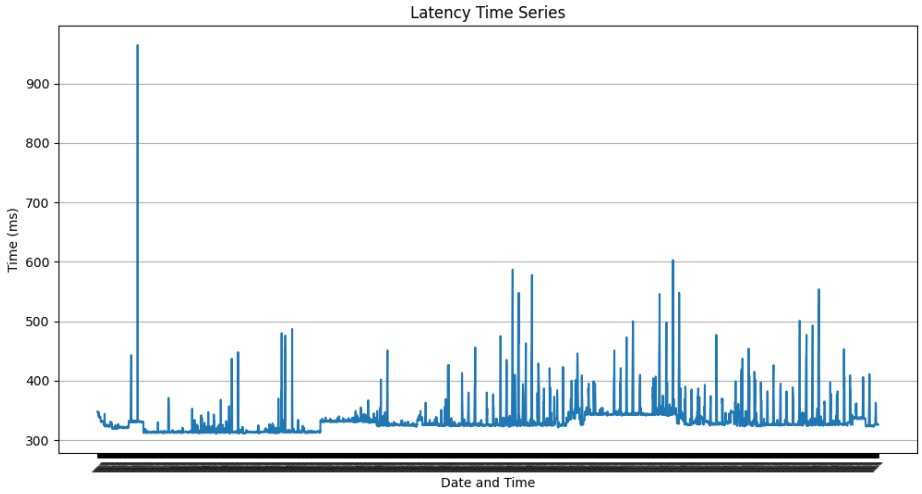
The chart shows very pronounced peaks, not only frequent but also very high, with some peaks even exceeding 700 milliseconds. Such frequent and high latency peaks suggest the server may have more serious network connection issues or limited processing capabilities.

AS1.i1:



The chart for shows that while latency remains at a lower level most of the time, there are occasional larger spikes, nearing 500 milliseconds at their highest. This indicates that the server maintains good performance most of the time but can occasionally be affected by network fluctuations or other external factors.

AS1.i2:



The chart displays even more extreme latency peaks, with these peaks nearing 900 milliseconds. Such high latency peaks are significant and may point to more severe network problems or stability issues with the server.

- Autocorrelation plot

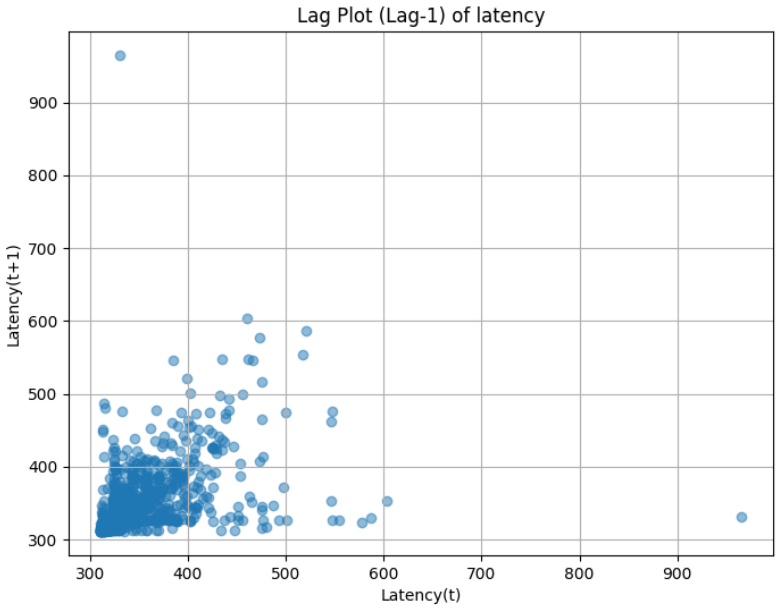
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import gaussian_kde
import pandas as pd


def extract_latencies(filename):
    # Initialize a list to store extracted latency values
    latency_values = []

    # Open the file and read each line
    with open(filename, 'r') as file:
        for line in file:
            if 'time=' in line:
                # Find lines containing 'time=' and split the string
                parts = line.split()
                for part in parts:
                    if part.startswith('time='):
```

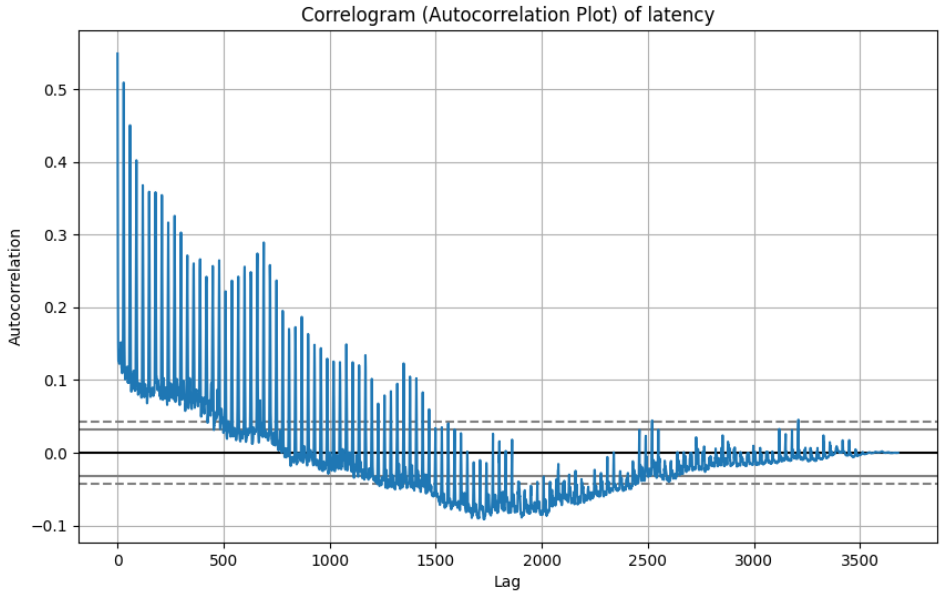
```
                    # Extract the latency value and remove 'ms'
                    latency = part.split('=')[1].rstrip(' ms')
                    try:
                        # Convert the extracted latency value to a
float and store it
                        latency_values.append(float(latency))
                    except ValueError:
                        # Skip the value if conversion fails
                        continue

    return latency_values


def create_lag_plot(latency_values):
    # Create a lag plot
    plt.figure(figsize=(8, 6))
    plt.scatter(latency_values[:-1], latency_values[1:], alpha=0.5)
    plt.title('Lag Plot (Lag-1) of latency')
    plt.xlabel('Latency(t)')
    plt.ylabel('Latency(t+1)')
    plt.grid(True)
    plt.show()


def create_autocorrelation_plot(latency_values):
    # Create an autocorrelation plot
    plt.figure(figsize=(10, 6))
    pd.plotting.autocorrelation_plot(latency_values, ax=plt.gca())
    plt.title('Correlogram (Autocorrelation Plot) of latency')
    plt.xlabel('Lag')
    plt.grid(True)
    plt.show()


if __name__ == "__main__":
    filename = 'files/ping/ok1.iperf.comnet-student.eu.txt'

    # Extract latency values from the file
    latency_values = extract_latencies(filename)
```

```
# Create and display the lag plot
create_lag_plot(latency_values)

# Create and display the autocorrelation plot
create_autocorrelation_plot(latency_values)
```

This code's functionality is to extract network latency data from a specified file and then use Matplotlib and Pandas to create two different plots for analyzing this latency data. First, it extracts the latency values from the file using the extract_latencies function and stores them in a list. Then, it creates a lag plot using the create_lag_plot function to visualize the relationship between latency data points. Next, it generates an autocorrelation plot using the create_autocorrelation_plot function to analyze the autocorrelation of the latency data.

AS1.i2:

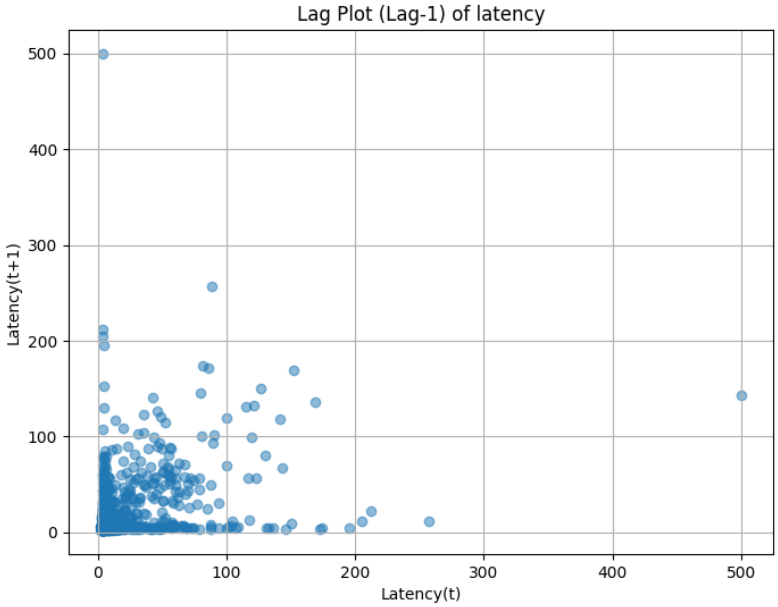


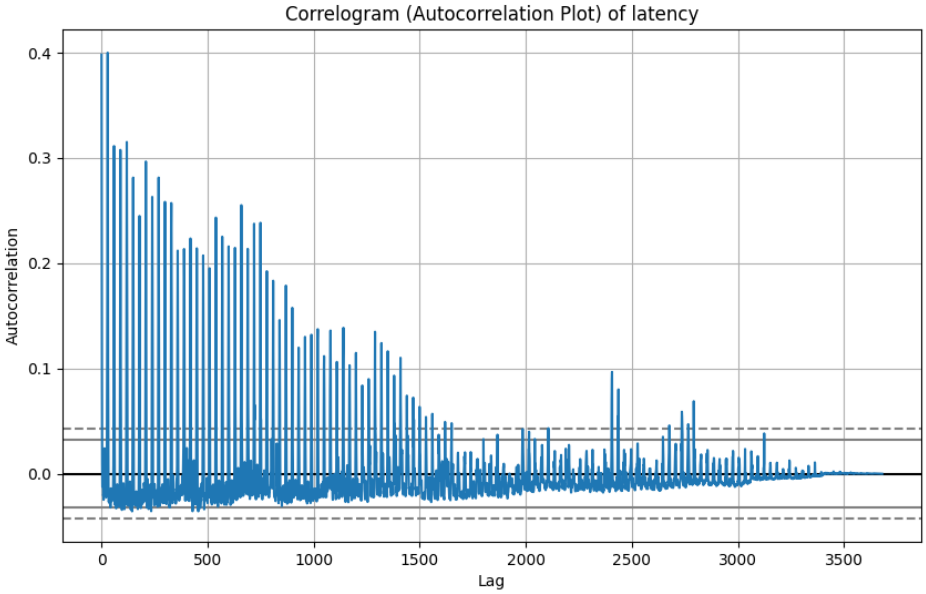


The lag plot shows the relationship between each observation and its previous one. The data points are concentrated in the lower left corner, indicating that at lower lag values, many observations have latency close to that of their previous observation. The absence of a clear structure or pattern suggests a random distribution of data points, which may imply that the values in the dataset are independent, with no obvious time series trend or periodicity.

The autocorrelation plot indicates the degree of autocorrelation of the data points, that is, the correlation of data points with their previous points in the time series. At initial lags (low lag values), the autocorrelation is higher, suggesting that observations in the time series are related to their immediate predecessors. As the lag increases, the autocorrelation gradually decreases, indicating that the association between observations diminishes over longer time intervals. The autocorrelation values drop below the confidence interval at certain points, suggesting that the correlations at these specific lags may not be coincidental.

AS1.i1:



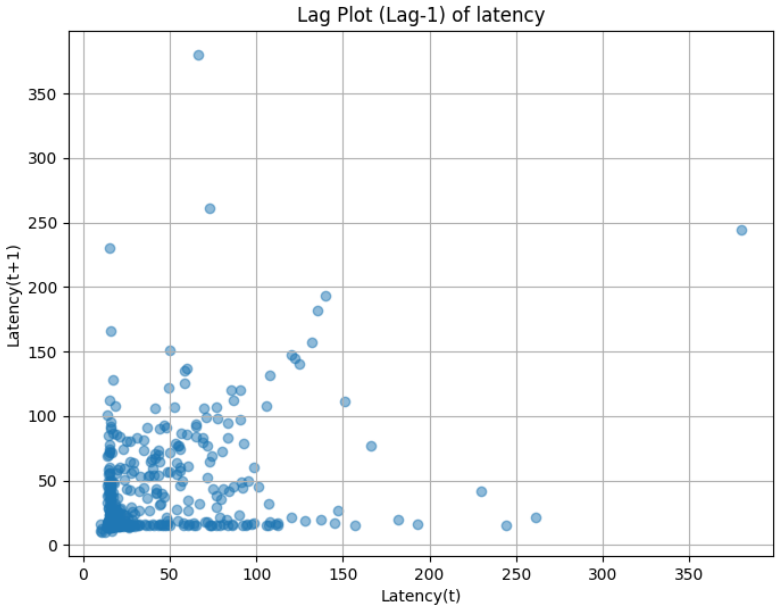


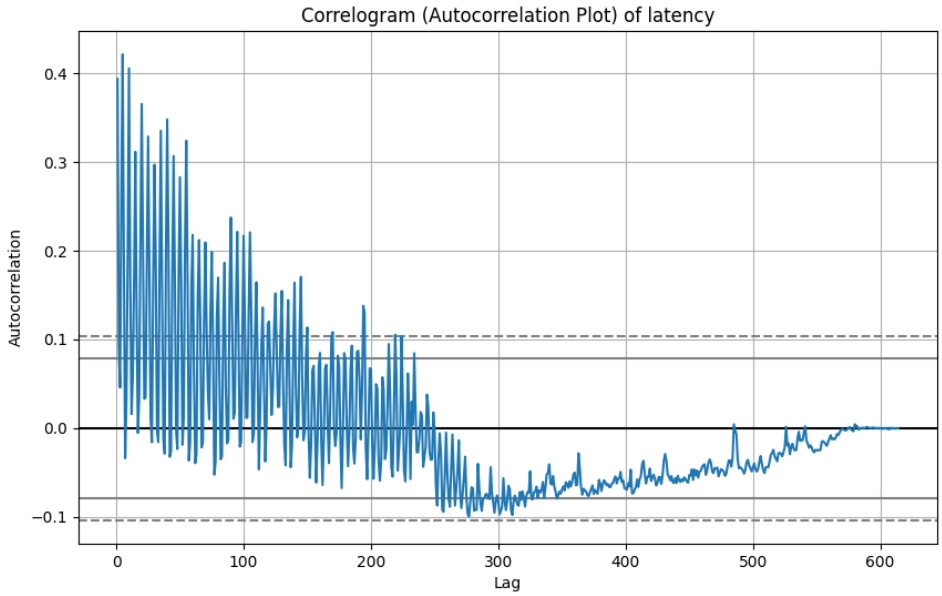
In the lag plot, data points are primarily concentrated near the origin, indicating that for most observations, the lag compared to the previous observation is quite small. As the lag values increase, the distribution of observations also broadens, but there is still no apparent pattern or trend, which may suggest that the values in the dataset are independent, without obvious trends or periodicity in the time series.

The correlogram displays the autocorrelation of the data, which decreases rapidly with increasing

lag. Initially, the autocorrelation is quite high, meaning that observations in the series are highly correlated with their preceding values. The autocorrelation drops off quickly as lag increases, tending towards zero after nearly 50 lag units, indicating that the values in the time series become rapidly uncorrelated as the lag increases. The peaks in the autocorrelation plot decrease swiftly and approach near zero, suggesting any periodic features in the data are very short-lived.

AS1.n1:





The lag plot shows the relationship at lag-1 between each observation and its preceding value. Data points are densely packed near the origin, indicating that many observations do not change much compared to their previous value. The absence of a clear pattern or trend in the plot typically suggests that the observations in the series may be randomly distributed, without showing strong autocorrelation or time dependency.

The correlogram displays the change in autocorrelation of the data with the lag. Autocorrelation is relatively high at initial lags, but as the lag increases, this correlation quickly diminishes. Autocorrelation stabilizes around 100 lag units and remains close to zero or slightly fluctuates in subsequent lags. Most of the autocorrelation values are within the confidence interval without any significant breaches, indicating that these correlations might be due to randomness rather than actual associations in the data.

3. Throughput

```
import re
import matplotlib.pyplot as plt
import numpy as np

# Rename variables for clarity
download_speeds = []
```

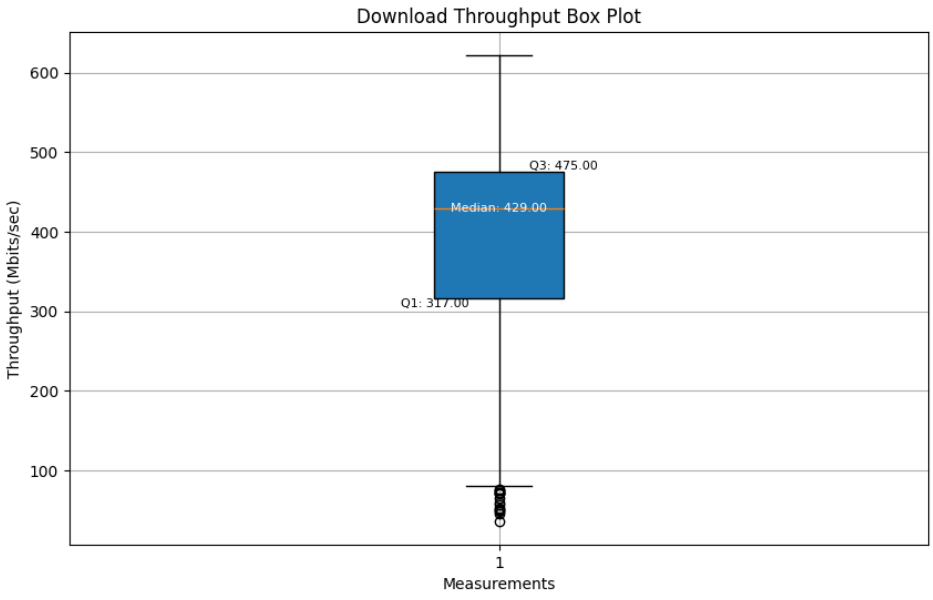
```
upload_speeds = []

# Process the file and collect download and upload speeds
with open('files/iperf/ok1.iperf.comnet-student.eu.txt', 'r') as
file:
    for line in file:
        stripped_line = line.strip()
        if re.search("receiver", stripped_line):
            items = stripped_line.split()
            download_speeds.append(float(items[6]))
            print(stripped_line)
        elif re.search("sender", stripped_line):
            items = stripped_line.split()
            upload_speeds.append(float(items[6]))
            print(stripped_line)

# Display download and upload speeds
print("Download Speeds:", download_speeds)
print("Upload Speeds:", upload_speeds)

# Create box plots for download speeds
fig, ax = plt.subplots(figsize=(10, 6))
boxplot_dict = ax.boxplot(download_speeds, patch_artist=True)

# Annotate median, quartiles, and labels
medians = [median.get_ydata()[0] for median in
boxplot_dict['medians']]
for tick, median in zip(ax.get_xticks(), medians):
    ax.text(tick, median, f'Median: {median:.2f}', ha='center',
va='center', fontdict={'fontsize': 8, 'color': 'white'})

boxes = [box.get_path().vertices for box in boxplot_dict['boxes']]
for box in boxes:
    box_bottom = box[0, 1]
    box_top = box[2, 1]
    ax.text(box[0, 0], box_bottom, f'Q1: {box_bottom:.2f}',
ha='center', va='top', fontdict={'fontsize': 8})
    ax.text(box[2, 0], box_top, f'Q3: {box_top:.2f}', ha='center',
va='bottom', fontdict={'fontsize': 8})
```

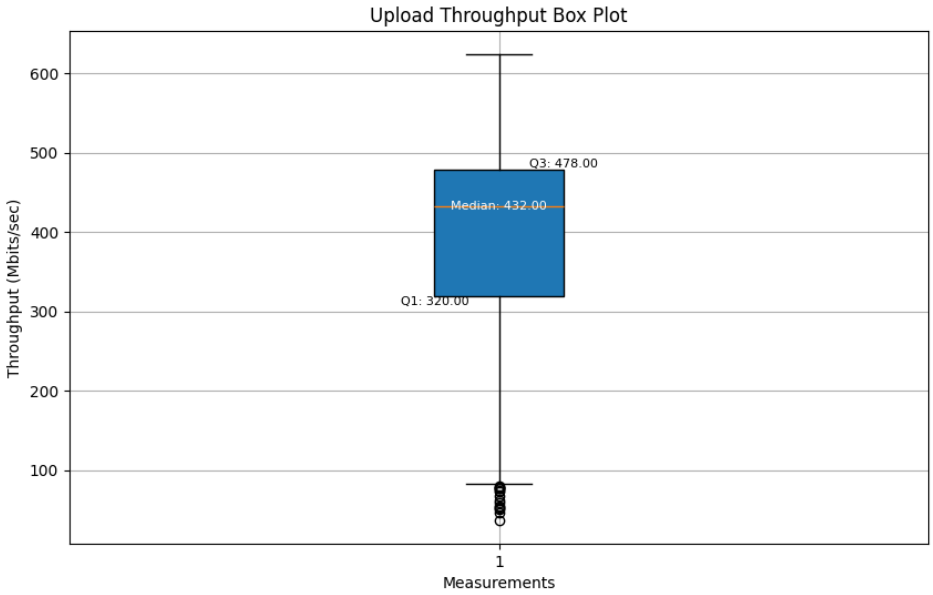
```
ax.set_title('Download Throughput Box Plot')
ax.set_ylabel('Throughput (Mbits/sec)')
ax.set_xlabel('Measurements')
plt.grid(True)
plt.show()

# Create box plots for upload speeds
fig, ax = plt.subplots(figsize=(10, 6))
boxplot_dict = ax.boxplot(upload_speeds, patch_artist=True)

medians = [median.get_ydata()[0] for median in
boxplot_dict['medians']]
for tick, median in zip(ax.get_xticks(), medians):
    ax.text(tick, median, f'Median: {median:.2f}', ha='center',
va='center', fontdict={'fontsize': 8, 'color': 'white'})

boxes = [box.get_path().vertices for box in boxplot_dict['boxes']]
for box in boxes:
    box_bottom = box[0, 1]
    box_top = box[2, 1]
    ax.text(box[0, 0], box_bottom, f'Q1: {box_bottom:.2f}',
ha='center', va='top', fontdict={'fontsize': 8})
    ax.text(box[2, 0], box_top, f'Q3: {box_top:.2f}', ha='center',
va='bottom', fontdict={'fontsize': 8})

ax.set_title('Upload Throughput Box Plot')
ax.set_ylabel('Throughput (Mbits/sec)')
ax.set_xlabel('Measurements')
plt.grid(True)
plt.show()

# Calculate statistics for download speeds
mean_download = np.mean(download_speeds)
harmonic_mean_download = 1 / np.mean(1 / np.array(download_speeds))
geometric_mean_download = np.exp(np.mean(np.log(download_speeds)))
median_download = np.median(download_speeds)

# Calculate statistics for upload speeds
mean_upload = np.mean(upload_speeds)
harmonic_mean_upload = 1 / np.mean(1 / np.array(upload_speeds))
```

```
geometric_mean_upload = np.exp(np.mean(np.log(upload_speeds)))
median_upload = np.median(upload_speeds)

# Display the statistics
print("Download Speeds Statistics:")
print("Mean:", mean_download)
print("Harmonic Mean:", harmonic_mean_download)
print("Geometric Mean:", geometric_mean_download)
print("Median:", median_download)

print("\nUpload Speeds Statistics:")
print("Mean:", mean_upload)
print("Harmonic Mean:", harmonic_mean_upload)
print("Geometric Mean:", geometric_mean_upload)
print("Median:", median_upload)
```

This code extracts network speed data from a text file, stores download and upload speeds in separate lists, then creates box plots to visualize the distribution of these speed data. Next, it calculates statistics for each speed list, including mean, harmonic mean, geometric mean, and median, and prints these statistical results for analysis and comparison of network performance.

AS1.i1:






```
Download Speeds Statistics:
Mean: 375.7710743801653
Harmonic Mean: 216.11100541704727
Geometric Mean: 313.9039944585448
Median: 429.0

Upload Speeds Statistics:
Mean: 378.4446280991736
Harmonic Mean: 222.1006817704386
Geometric Mean: 317.8221995381174
Median: 432.0
```

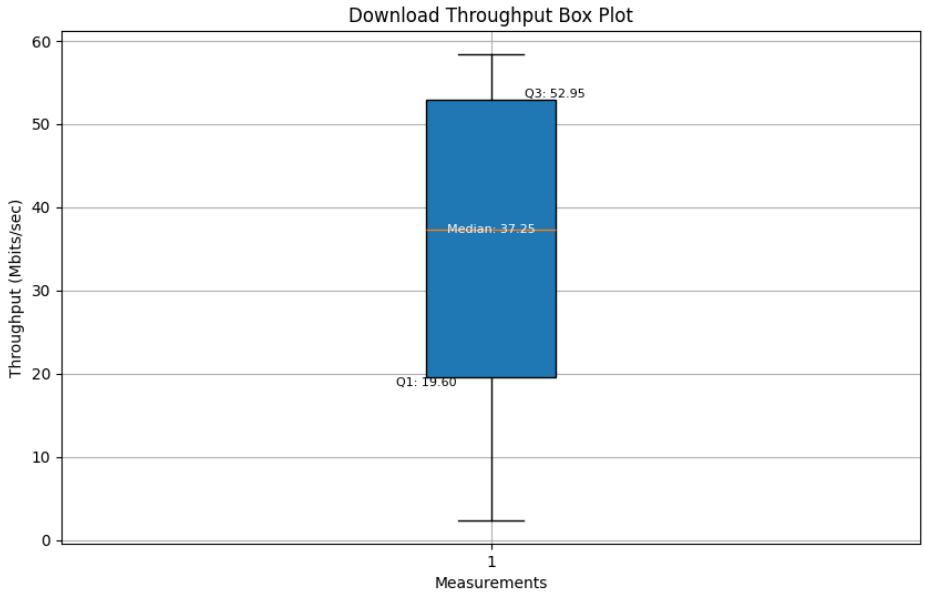
The download throughput box plot indicates that the median download speed is 429 Mbps. The first quartile (Q1) is 317 Mbps, showing that 25% of the download speed measurements are below this value. The third quartile (Q3) is 475 Mbps, meaning that 75% of download speeds are below this value. The range between Q1 and Q3, known as the interquartile range (IQR), represents the middle 50% of the data distribution. There are some outliers below the lower edge, indicating that some download speeds are significantly lower than the bulk of the data.

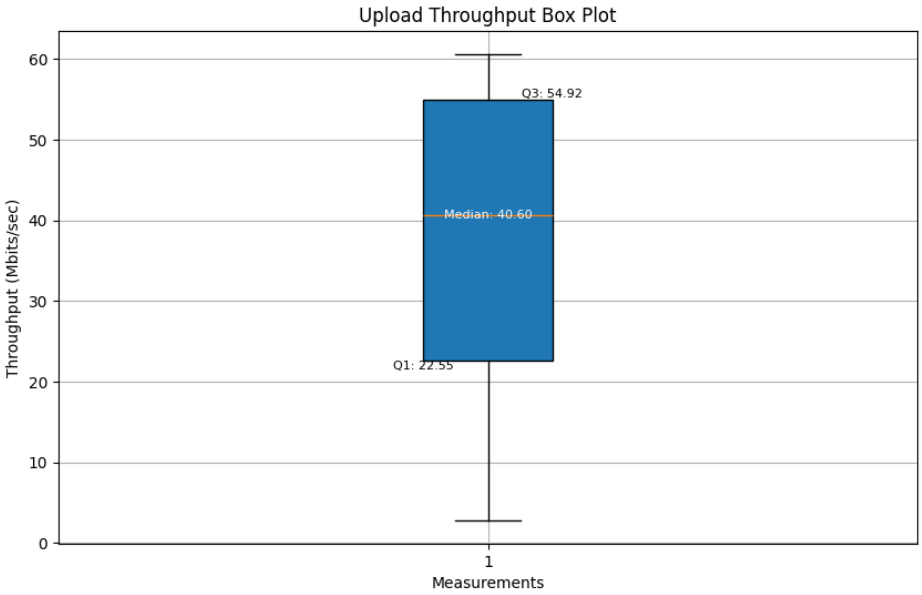
The upload throughput box plot shows a median upload speed of 432 Mbps. The first quartile (Q1)

is 320 Mbps, so 25% of upload speed measurements are below this value. The third quartile (Q3) is 478 Mbps, indicating that 75% of upload speeds are below this value. The IQR for upload speeds is slightly larger than that for download speeds, suggesting a slightly greater variability in the middle 50% of the upload data. Similar to download speeds, there are outliers below the lower edge, pointing out instances of significantly lower than normal upload speeds.

Overall, both download and upload speeds exhibit a right-skewed distribution, which is evident from the lower harmonic and geometric means compared to the arithmetic means. The median values are relatively high, indicating that more than half of the measurements are at the higher end of the throughput range. The presence of outliers, especially on the lower end, impacts the mean values, pulling them down, which is why the harmonic and geometric means are notably lower. The consistency of the median values in the box plots and the statistical data indicates an accurate representation of the visual data.

AS1.i2:






```
Download Speeds Statistics:
Mean: 34.305454545454545
Harmonic Mean: 13.307485148452628
Geometric Mean: 24.904824514956122
Median: 37.25

Upload Speeds Statistics:
Mean: 36.440000000000005
Harmonic Mean: 15.97553549810862
Geometric Mean: 27.530567037287646
Median: 40.6
```

Similarly, for AS2.i2, we can draw the following conclusions. The harmonic and geometric means being lower than the arithmetic means indicate that both download and upload speeds display a right-skewed distribution. The median values are higher than the arithmetic means, which is typical for a skewed distribution and confirms that the majority of speeds are above the average. The absence of outliers in both box plots suggests that there are no extreme values far from the bulk of the measurements. The median values in the statistical data closely align with those presented in the box plots, confirming the accuracy of the visual representation.

4. Throughput time series

```
import matplotlib.pyplot as plt
import re

def read_iperf_data(file_path):
    with open(file_path, 'r') as file:
        data = file.read()
    return data

def parse_iperf_data(data):
    timestamps = []
    speeds_dl = []
    speeds_ul = []
    last_is_date = False

    lines = data.split('\n')

    for line in lines:
        if re.search(r'\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}', line):
            if not last_is_date:
                timestamps.append(line.strip())
                last_is_date = True
            else:
                timestamps.pop()
                timestamps.append(line.strip())
        else:
            striped_line = line.strip()
            if re.search("receiver", striped_line):
                items = striped_line.split()
                speeds_dl.append(float(items[6]))
                last_is_date = False
            elif re.search("sender", striped_line):
                items = striped_line.split()
                speeds_ul.append(float(items[6]))
                last_is_date = False

    return timestamps, speeds_ul, speeds_dl

def plot_iperf_data(timestamps, speeds_ul, speeds_dl):
```

```
    plt.figure(figsize=(12, 6))
    plt.plot(timestamps, speeds_ul, label='UL Speed', marker='o')
    plt.plot(timestamps, speeds_dl, label='DL Speed', marker='x')
    plt.title('Time Series of Sender and Receiver Speed')
    plt.xlabel('Timestamp')
    plt.ylabel('Speed (Mbits/sec)')
    plt.legend()
    plt.grid(True)
    plt.xticks(rotation=90, fontsize=5)
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    file_path = 'files/iperf/ok1.iperf.comnet-student.eu.txt'
    iperf_data = read_iperf_data(file_path)
    timestamps, speeds_ul, speeds_dl = parse_iperf_data(iperf_data)
    plot_iperf_data(timestamps, speeds_ul, speeds_dl)
```

This code first reads network performance test data from a specified file using iperf. It then parses the data to extract timestamps, upload speeds, and download speeds. Next, it utilizes the Matplotlib library to create a time series chart, where the x-axis represents timestamps and the y-axis represents speeds (both upload and download). Upload and download speeds are labeled on the chart.

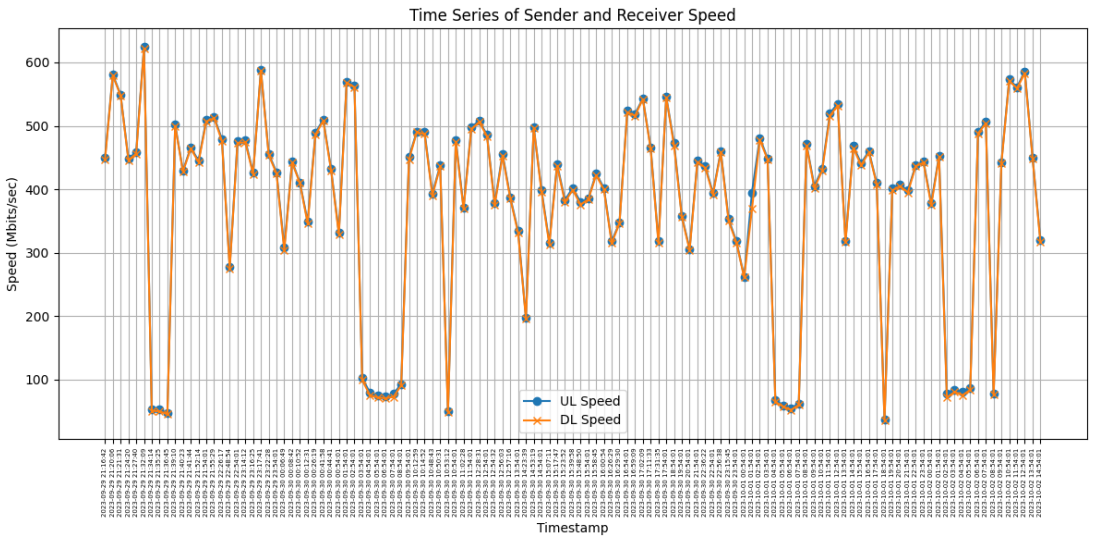
```
import matplotlib.pyplot as plt
import re
import pandas as pd

# Read data from the file
with open('files/iperf/ok1.iperf.comnet-student.eu.txt', 'r') as
file:
    data = file.read()

# Extract time and speed data
timestamps = []
speeds_dl = []
speeds_ul = []

# Initialize a variable to track the last line type (0: none, 1:
date, 2: receiver, 3: sender)
```

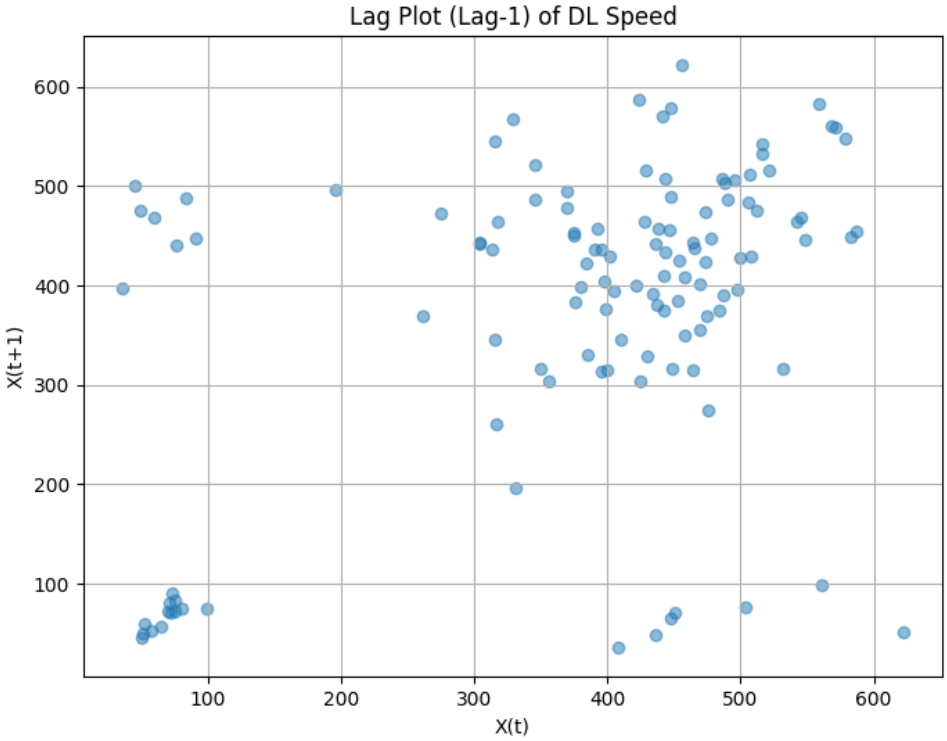
```
last_line_type = 0

lines = data.split('\n')

for line in lines:
    if re.search(r'\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}', line):
        if last_line_type == 1:
            timestamps.pop()
        timestamps.append(line.strip())
        last_line_type = 1
    else:
        striped_line = line.strip()
        if re.search("receiver", striped_line):
            items = striped_line.split()
            speeds_dl.append(float(items[6]))
            last_line_type = 2
        elif re.search("sender", striped_line):
            items = striped_line.split()
            speeds_ul.append(float(items[6]))
            last_line_type = 3

# Create lag plots and autocorrelation plots for download speed (DL)
plt.figure(figsize=(8, 6))
plt.scatter(speeds_dl[:-1], speeds_dl[1:], alpha=0.5)
plt.title('Lag Plot (Lag-1) of DL Speed')
plt.xlabel('X(t)')
plt.ylabel('X(t+1)')
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 6))
pd.plotting.autocorrelation_plot(speeds_dl, ax=plt.gca())
plt.title('Correlogram (Autocorrelation Plot) of DL Speed')
plt.xlabel('Lag')
plt.grid(True)
plt.show()

# Create lag plots and autocorrelation plots for upload speed (UL)
plt.figure(figsize=(8, 6))
plt.scatter(speeds_ul[:-1], speeds_ul[1:], alpha=0.5)
```

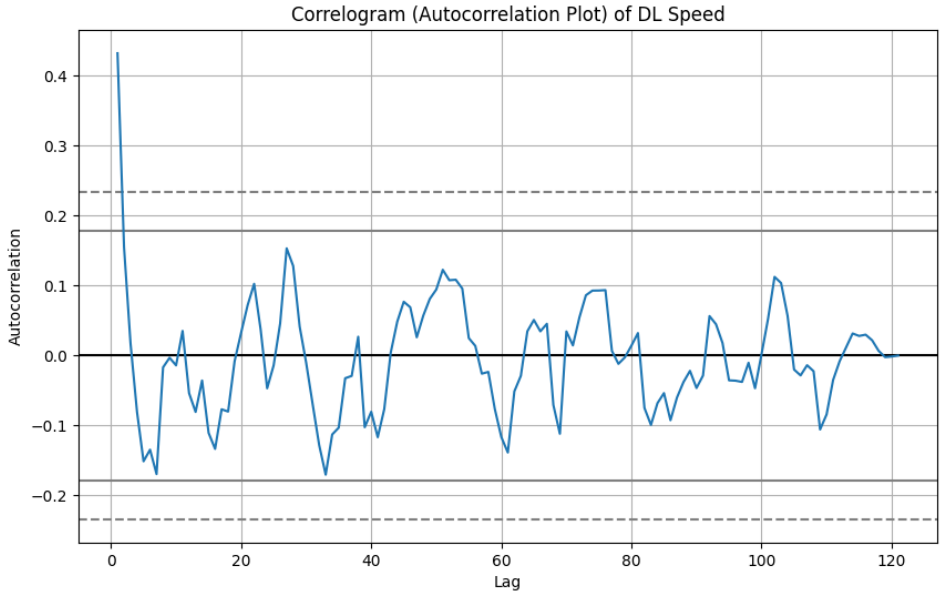
```
plt.title('Lag Plot (Lag-1) of UL Speed')
plt.xlabel('X(t)')
plt.ylabel('X(t+1)')
plt.grid(True)
plt.show()

plt.figure(figsize=(10, 6))
pd.plotting.autocorrelation_plot(speeds_ul, ax=plt.gca())
plt.title('Correlogram (Autocorrelation Plot) of UL Speed')
plt.xlabel('Lag')
plt.grid(True)
plt.show()
```

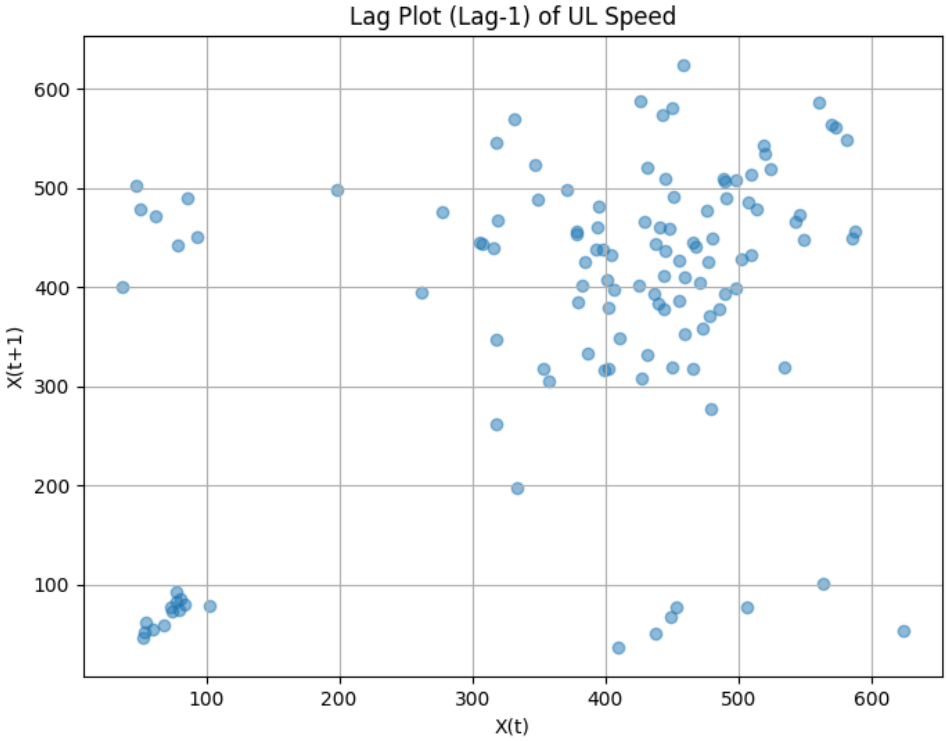
The purpose of this code is to read network performance test data from a specified file using iperf, extract timestamps, download speeds, and upload speeds from the data, and create lag plots and autocorrelation plots. These plots help analyze the relationship between download speeds and upload speeds as well as their autocorrelation over time.

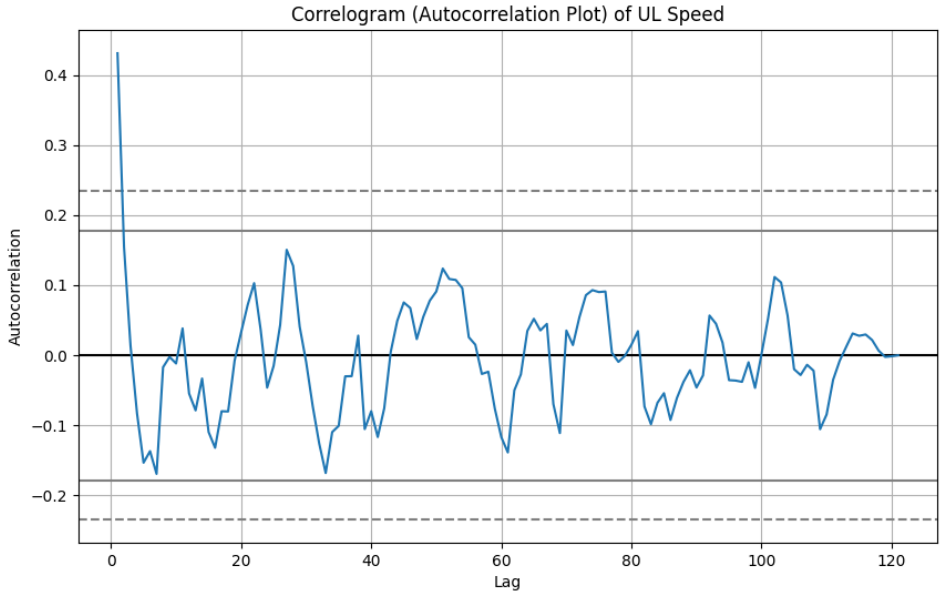
AS2.i1:




Lag Plot (Lag-1) of DL Speed
X(t+1)
X(t)
100
200
300
400
500
600


Correlogram (Autocorrelation Plot) of DL Speed
Autocorrelation
Lag
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
0
20
40
60
80
100
120


Lag Plot (Lag-1) of UL Speed
X(t+1)
X(t)
100
200
300
400
500
600

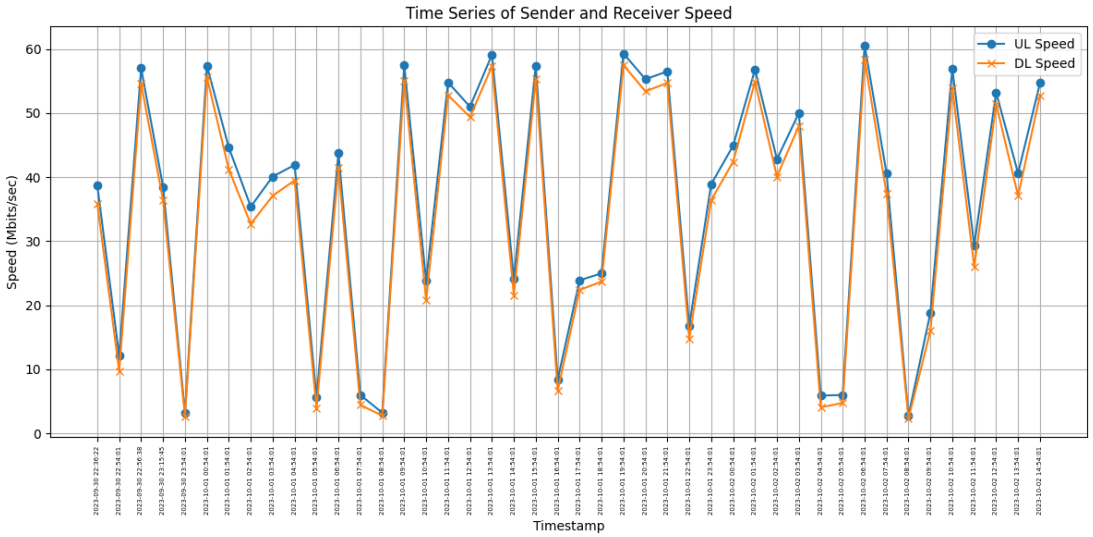

Correlogram (Autocorrelation Plot) of UL Speed
Autocorrelation
Lag
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
0
20
40
60
80
100
120

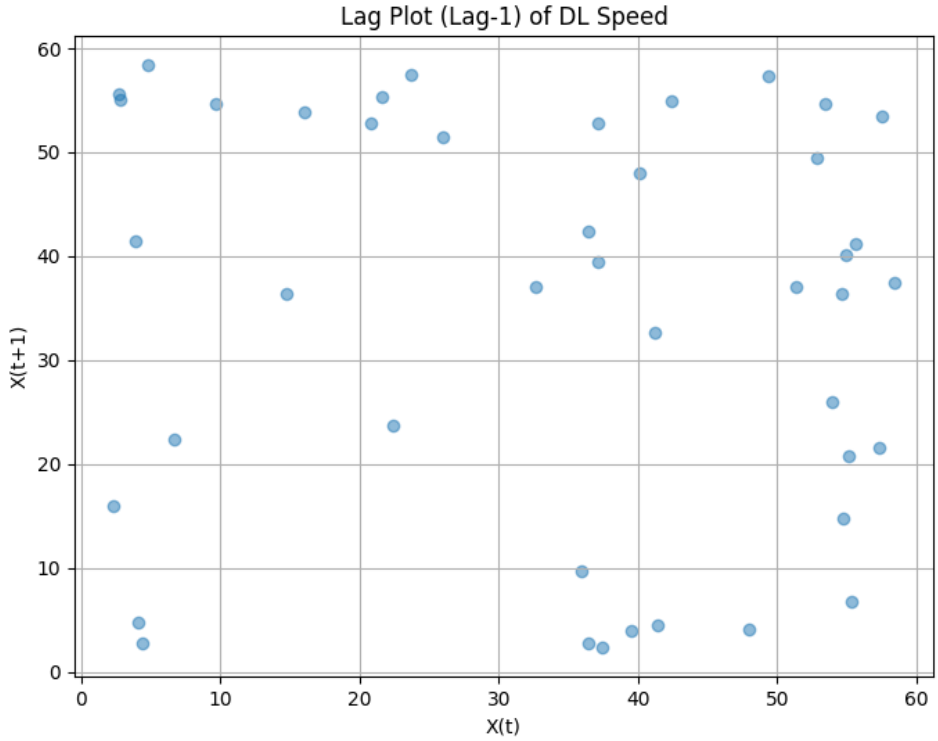
The first graph displays the time series data, where both upload speed (UL Speed) and download speed (DL Speed) vary over time. We can observe that both demonstrate significant fluctuations, indicating that at different points in time, there are notable peaks and troughs in speed. There seems to be a pattern of periodicity in both upload and download speeds, suggesting the possibility of periodic patterns in network usage, such as peak and off-peak periods.

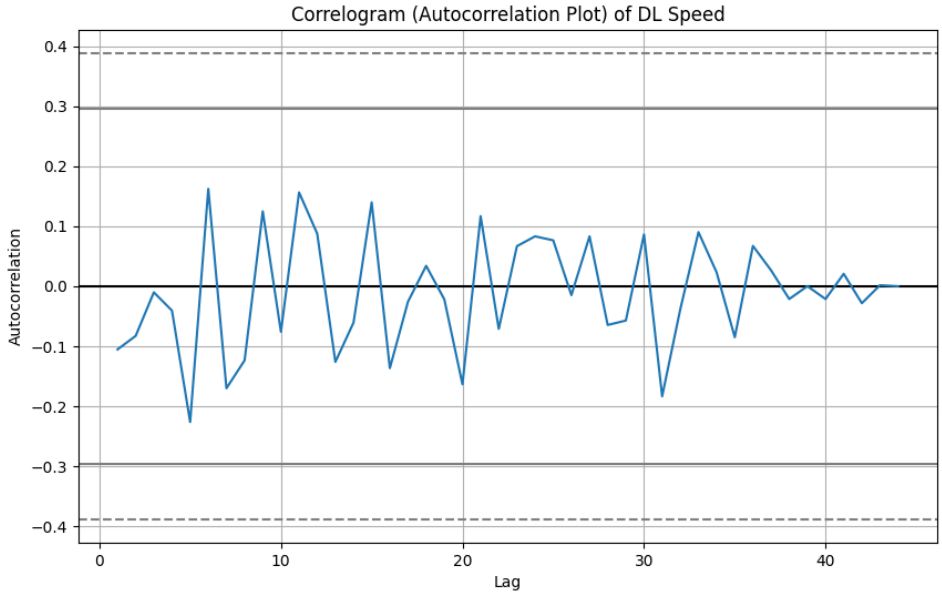
The second and fourth graphs are lag plots for upload and download speeds. These plots show the relationship of data points with their previous time point (lag-1). For the download speed (DL Speed), the lag plot exhibits a degree of positive linear relationship, indicating that the value at one time point is related to its previous value, which may imply autocorrelation. For the upload speed (UL Speed), this relationship seems to be less pronounced, but a degree of positive linear relationship can still be observed.

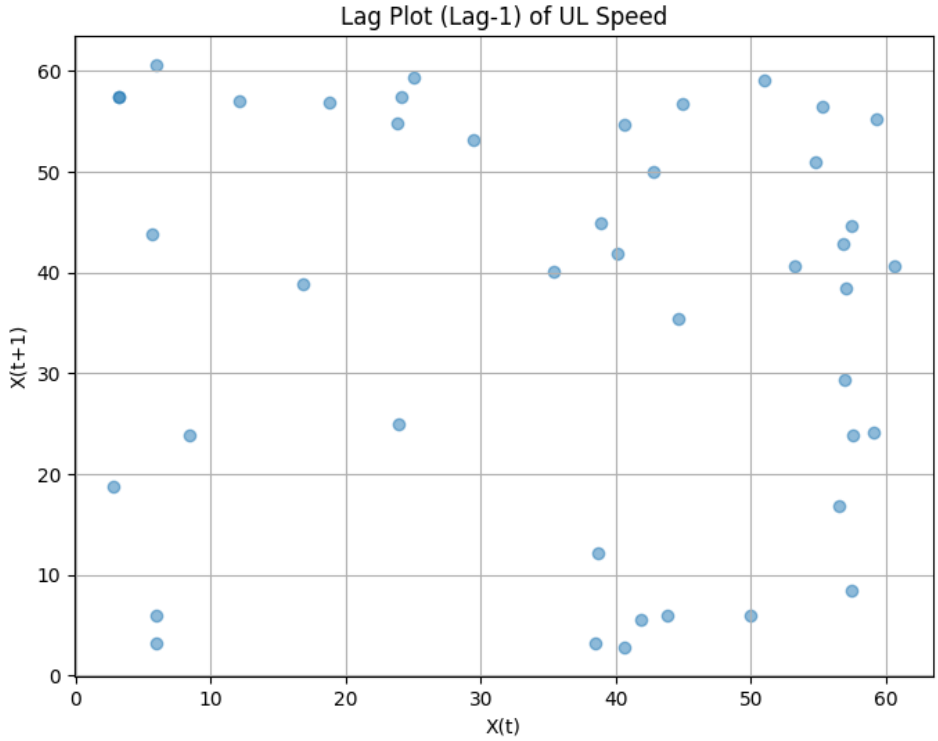
The third and fifth graphs are autocorrelation plots of upload and download speeds, showing the degree of autocorrelation at different lags. For download speed, the autocorrelation plot reveals a high initial autocorrelation at the first lag, but this correlation quickly decreases and fluctuates at different lags. This indicates the possibility of short-term dependencies. For upload speed, the autocorrelation plot also shows a high initial autocorrelation, but this correlation weakens as lag increases and exhibits greater volatility, which might indicate the presence of noise in the data.

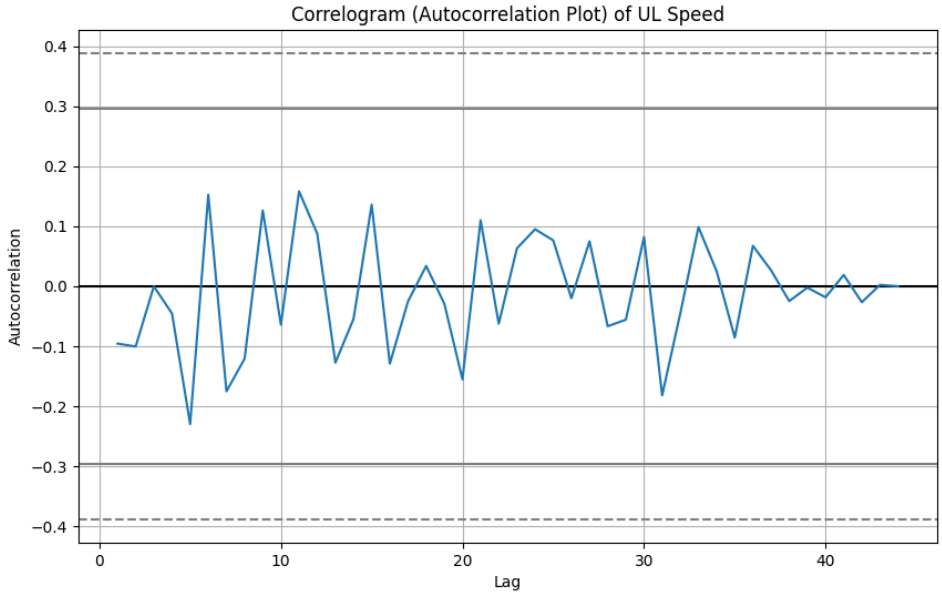
AS2.i2:




Lag Plot (Lag-1) of DL Speed
X(t+1)
X(t)
0
10
20
30
40
50
60


Correlogram (Autocorrelation Plot) of DL Speed
Autocorrelation
Lag
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
0
10
20
30
40


Lag Plot (Lag-1) of UL Speed
X(t+1)
X(t)
0
10
20
30
40
50
60


Correlogram (Autocorrelation Plot) of UL Speed
Autocorrelation
Lag
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.4
0
10
20
30
40

The first graph displays the time series of upload speed (UL Speed) and download speed (DL Speed). There is a certain level of variability over time, but the amplitude of fluctuations is much smaller compared to the AS2.i1 dataset. This suggests more stable network performance, with relatively consistent network conditions during the test period. There is a certain synchronicity between upload and download speeds, with both increasing or decreasing simultaneously at most time points, potentially indicating that they are affected by similar network conditions.

The second graph is a lag plot for download speed (DL Speed). Compared to AS2.i1, this dataset's lag plot shows a more dispersed pattern, indicating a less pronounced association between a speed measurement and its preceding value. The fourth graph, the lag plot for upload speed (UL Speed), also displays dispersed points, suggesting a similarly weak temporal association for upload speeds.

The third graph is an autocorrelation plot for download speed (DL Speed). It shows the autocorrelation coefficients fluctuating at different lags but generally trending near zero, which indicates a lack of strong autocorrelation between observations of download speed. The fifth graph is an autocorrelation plot for upload speed (UL Speed). The fluctuation pattern is similar to that of download speed, with autocorrelation coefficients primarily oscillating around the zero line, indicating that the autocorrelation between observations of upload speed is also not strong.

Compare to 3.2:

The AS1 series exhibits stronger autocorrelation during the initial lags and maintains a higher level of significance over a larger number of lags, suggesting that the AS1 series might have more predictable patterns or trends.

The AS2 series shows less persistence in both download and upload speeds, with autocorrelation quickly dropping to near-zero levels after a few initial lags. This may imply that the AS2 series is more influenced by random variations or external factors that disrupt the signal's persistence. In both AS2.i1 and AS2.i2, download and upload speeds exhibit similar behaviors, suggesting that the underlying factors affecting download and upload speeds are likely similar within each dataset.

Overall, the AS1 datasets seem to represent processes with longer memories and more predictable patterns, while the AS2 datasets are characterized by lower predictability and greater randomness over time.

- Conclusion

  - Describe the system from which you obtained measurements and any challenges you encountered during the process.

    I encountered the following challenges. First, collecting data with sufficient granularity to capture fluctuations without creating an overwhelming volume can be challenging. Balancing granularity, storage capacity, and processing power is often a key issue. Network performance can vary due to a multitude of factors such as bandwidth, network traffic, hardware limitations, and service provider issues. This variability makes it difficult to establish a baseline for performance. Ensuring the accuracy and consistency of the collected data can be difficult, especially if the measurements are affected by transient issues that do not reflect the typical performance of the network. External factors, such as server response times, routing paths, and end-user device performance, can also affect the measurements.

  - Was there any correlation between the path length (number of routers, which can be checked using traceroute and/or the TTL value of ICMP Echo Responses) and the stability of measurements? If you also recorded the TTL value, did it change over time?

    Longer network paths, indicated by a greater number of routers or hops, can potentially lead to decreased stability in measurements. Each additional router introduces a new potential point for delays, packet loss, or variations. However, this relationship is not always direct. Network stability also depends on the performance and reliability of each router in the path, overall network traffic, and the quality of the network links.

    TTL values can indicate the length of a network path, as each router along the path typically decrements the TTL by one. However, the starting TTL value set by the sending host can vary, so the absolute TTL value may not always accurately reflect the exact number of hops. Changes in TTL values over time can indicate changes in the network path. For example, if the network dynamically reroutes traffic due to congestion or link failure, this might be reflected in varying TTL values.

    If TTL values change frequently, it could indicate a less stable network path, as route changes could affect latency and packet loss. Conversely, consistent TTL values over time

might suggest a stable route without frequent path changes, which could correlate with more stable network performance measurements.

- Was there any observable correlation between the throughput and latency?

  In general, there is an inverse relationship between throughput and latency. High latency typically leads to lower throughput, and vice versa. This is because high latency may indicate delays in the transmission of data packets, which can consequently reduce the rate at which data is successfully transferred (throughput). In networks with high latency, even with high bandwidth, throughput may not be fully utilized due to the delay in acknowledging received packets. During network congestion, both latency and throughput can be adversely affected. Latency increases because packets take longer to be processed and delivered, and throughput decreases due to packet loss and the need for retransmissions.

- Final conclusions

  - How was your own traffic (Task 1) different from the data provided (Task 2)? What kind of differences can you identify? What could be a reason for that?

    In my own traffic (Task 1), it can be observed that the destination ports are more concentrated on commonly used ports, such as HTTPS (port 443). The port distribution is relatively stable because I only have access to a limited variety of services. Additionally, the overall traffic is low, but there may be sudden spikes. Traffic patterns may fluctuate significantly over time, reflecting individual usage habits.

    On the other hand, the provided data (Task 2) includes a wider range of destination ports, encompassing various enterprise software, applications, mail servers, databases, internal web applications, and various remote access services. Traffic is higher during working hours and may be lower during non-working hours. Overall traffic is higher and exhibits a more even distribution, reflecting day-to-day business activities.

  - Comparing RTT latency about TCP connections (3.1), were active latency measurements around the same magnitude or was another much larger than the other?

    The active latency measurements vary across different orders of magnitude. Some servers exhibit very low latency, consistently staying below 20 milliseconds, while others have latencies around 50 milliseconds. There are also servers with latencies ranging from 200 milliseconds to 300 milliseconds. Therefore, the active latency measurements span different orders of magnitude and are not clustered around the same magnitude.

  - Discuss how data protection needs to be taken into account if you as a network provider employee were doing similar measurements as in this assignment in a network provider network (traffic generated by customers that may be private persons or companies).

    If I were an employee of a network provider conducting similar measurements in a network provider network where traffic is generated by both individuals and companies, data protection would be a significant consideration.

1) Privacy Concerns: Whether it's individuals or companies, customers using the services of a network provider have a reasonable expectation of privacy. Any data collected during measurements should be anonymized and aggregated to ensure that individual users or businesses cannot be identified from the collected data.
2) Data Minimization: Data collection should be limited to only what is strictly necessary for the specific measurements or analysis. Avoid collecting any unnecessary personal or sensitive information.
3) Informed Consent: Ensure that customers are informed about the nature and purpose of the measurements being conducted. If feasible, customers should have the option to choose to participate in or opt out of data collection.
4) Data Security: Implement robust security measures to prevent unauthorized access, breaches, or data leaks. This includes the use of encryption, access controls, and other security protocols to protect the data.
5) Data Retention: Define clear policies regarding data retention and deletion. Data should not be retained longer than necessary for the intended purpose, and there should be a secure process for disposing of data when it's no longer needed.
6) Compliance with Regulations: Comply with relevant data protection regulations and laws, such as the European GDPR (General Data Protection Regulation).
7) Transparency: Maintain transparency about data collection and processing practices. Customers should have access to information about what data is being collected, how it's used, and who can access it.
8) Data Use Limitation: Ensure that the collected data is used only for the specific purpose of network performance measurements and analysis. It should not be repurposed for other uses without explicit consent.
9) Data Access Requests: Be prepared to respond to customer requests for access to their own data or requests for data deletion in accordance with applicable data protection laws.
10) Regular Audits: Conduct regular internal audits and assessments of data protection practices to identify and address potential risks and compliance issues.

In summary, when conducting measurements in a network provider network with data generated by customers, data protection and privacy considerations should be a top priority. Adhering to privacy principles, regulations, and best practices is essential for maintaining customer trust and complying with legal requirements.

- Discuss how data protection needs to be taken into account if you as a company ICT support group employee were doing similar measurements as in this assignment in a company network (traffic generated by employees and customers).

1) Ownership of Data and Consent: Determine who owns the data being collected, whether it's the company, its employees, or its customers. Obtain explicit data collection consent from employees and customers in accordance with data ownership and local regulations, especially if data collection extends beyond regular network monitoring.
2) Anonymization and Pseudonymization: Ensure that any personal or sensitive data collected is subjected to anonymization to prevent the identification of individual employees or customers. If complete anonymization is not feasible, pseudonymization can be employed to replace personally identifiable information (PII) with artificial identifiers.
3) Data Minimization: Collect only the data that is strictly necessary for network performance measurement and analysis. Avoid collecting any unnecessary personal or sensitive information that is not directly related to the measurement objectives.
4) Security Measures: Utilize strong encryption to safeguard data during transit and storage, preventing unauthorized access or breaches. Implement access controls to ensure that only authorized personnel can access the collected data. Conduct regular security audits to identify vulnerabilities and ensure compliance with data protection policies.
5) Data Retention and Deletion: Define explicit data retention policies specifying how long data will be stored. Establish secure data disposal procedures to prevent data leaks or violations when data is no longer needed.
6) Transparency and Communication: Clearly communicate to employees and customers the information pertaining to the data being collected, the reasons for its collection, and how it will be used. Provide easily accessible privacy policies outlining data handling practices.
7) Compliance with Regulations: Ensure that data collection and processing practices adhere to applicable data protection regulations and laws, such as GDPR, CCPA, or other regional laws.

- How do you rate the complexity of different tasks? Were some tasks more difficult or laborious than others? Did data volume cause any issues with your analysis?

  For me, both Task 2 and Task 3 are considered complex. In Task 2, I need to delve into a lot of advanced computer knowledge and write a significant amount of scripts. In Task 3, I have to switch between many different files, which also results in a substantial workload. In contrast, Task 1 is relatively straightforward as it involves simply observing and describing the data. Considering data volume, while dealing with larger datasets may require more time, it typically doesn't pose issues as long as memory and computing resources are managed effectively. Therefore, data volume might extend the time required for the tasks but doesn't necessarily make them more complex. Overall, I find the tasks in the final assignment to be quite demanding and time-consuming.