BANDWIT: Visualization Tool for Hop-by-Hop Delay and Bandwidth Analytics

Heer Kubadia 22110096
IIT Gandhinagar
heer.kubadia@iitgn.ac.in

Lavanya 22110130
IIT Gandhinagar
lavanya.lavanya@iitgn.ac.in

Anura Mantri 22110144
IIT Gandhinagar
anura.mantri@iitgn.ac.in

Neerja Kasture 22110165
IIT Gandhinagar
neerja.kasture@iitgn.ac.in

***Abstract*—This report presents the design and implementation of *BANDWIT* - a network performance analysis and visualization tool that provides hop-by-hop delay and bandwidth metrics using ICMP and UDP probing techniques. The system also features a web-based interface to execute the traceroute functionality, generate interactive visualizations, assist in identifying bottlenecks within the network and and generates detailed network reports, highlighting hop-by-hop delays and capacity limitations. The backend is developed in C++ for precise control using raw sockets, while the frontend is powered by Python using FastAPI. Results are visualized using interactive maps and graph-based topology diagrams. Our study also demonstrates the tradeoffs of ICMP and UDP for bandwidth-aware network diagnosis.**

***Index Terms*—Network Performance, Traceroute, Bandwidth Estimation, RTT, Bottleneck Detection, FastAPI, Visualization, ICMP, UDP**

I. Introduction

Modern networked systems depend heavily on reliable, high-performance communication. Monitoring network paths for latency and throughput is crucial for diagnosing performance bottlenecks, identifying misconfigurations, and improving system responsiveness. Traditional tools like 'traceroute' provide visibility into the sequence of routers a packet traverses, but are limited to round-trip time (RTT) analysis and lack bandwidth or congestion insights. Also, ICMP-based methods can sometimes be inaccurate or inefficient due to rate limiting and low prioritization on routers.

To address these limitations, we designed and implemented **BANDWIT**, a hop-by-hop network analysis tool. BANDWIT enhances traditional tools by incorporating both ICMP and UDP-based probing mechanisms. It measures per-hop RTT, estimates bandwidth using packet pair dispersion, and visualizes the end-to-end path using an interactive graph. By combining low-level packet engineering with modern visualization techniques, our tool helps users understand how data flows across the network—and where it may slow down.

II. Literature Review

The purpose of this literature review is to explore existing tools and methodologies for network path analysis and per-hop bandwidth estimation. By analyzing both open-source and commercial software, we aim to identify common practices, evaluate limitations, and determine requirements for building a lightweight, accurate, and user-friendly tool for bandwidth estimation.

A. Why the Literature Review Was Conducted

This review was conducted to understand:

- The capabilities of current tools in estimating per-hop bandwidth and latency.
- Techniques used for network probing and route analysis.
- Shortcomings in tools that offer real-time and detailed per-hop bandwidth insights.
- Interface, performance, and deployment characteristics of various commonly used platforms.

B. Open Source Tools

Several open-source tools play a significant role in the ecosystem of network monitoring and analysis:

- **Zabbix**: A widely-used infrastructure monitoring solution that collects time-series data from devices, servers, and networks. Though robust, it lacks fine-grained per-hop analysis features.
- **OpenNMS**: An enterprise-grade open-source platform focused on network fault and performance management. It supports SNMP-based traffic monitoring but not detailed bandwidth estimation per hop.
- **Nagios**: A popular network monitoring tool with a plugin-based architecture. It provides basic network checks and alerts but does not natively support bandwidth probing or path discovery.
- **pchar** and **pathchar**: These command-line tools estimate per-hop latency and bandwidth using varying packet sizes and delay measurements. They are academically significant but require manual interpretation and lack modern visualization support.
- **Paris traceroute**: An improvement over the classic traceroute, this tool ensures consistent route mapping in load-balanced networks. Although it does not estimate bandwidth, it contributes accurate path discovery which is critical for follow-up analysis.
- **iperf3**, **bwping**: These tools provide end-to-end bandwidth testing but do not break down performance per hop. They are useful for assessing link capacity between two endpoints.
- **MyTraceroute (mtr)**: A dynamic network diagnostic tool that combines the functionality of traceroute and ping. It continuously sends probes to each hop and displays updated latency and packet loss statistics in real

time. While it does not compute bandwidth directly, its output and route stability information are valuable for identifying network bottlenecks.

C. Commercial Tools

Several commercial tools expand upon the capabilities of open-source solutions, offering enterprise features such as centralized control, rich visualizations, and SLA tracking:

- **Obkio**: A performance monitoring tool that tracks end-to-end and per-hop metrics such as latency, jitter, and bandwidth using lightweight agents. It is useful for detecting intermittent or location-specific performance issues.
- **NetScout**, **ThousandEyes**, and **SolarWinds**: These commercial-grade platforms provide full-stack network performance monitoring, root cause analysis, and in-depth visualization tools. However, they may be cost-prohibitive and are less customizable than open-source alternatives.
- **PingPlotter**:A network monitoring tool that visualizes latency, packet loss, and route behavior over time. It provides continuous traceroute-style probing with charts to identify issues and path stability. While it doesn't estimate per-hop bandwidth, it's easy to use and valuable for both network admins and non-technical users.

D. Insights Gained from the Literature

- Open-source tools tend to favor extensibility and community support, though they often require technical expertise and manual setup.
- Tools like `pchar` and `pathchar` attempt bandwidth estimation, but are not widely integrated into modern monitoring pipelines.
- `mtr` and `Paris traceroute` enhance route tracing, which can complement bandwidth tools.
- Commercial tools offer better usability and automation but may not allow low-level customization or detailed probing techniques.
- There is an opportunity to bridge the gap between raw bandwidth estimation and real-time, user-friendly diagnostics.

The literature review highlights that while there are many tools for general network monitoring and path discovery, very few focus specifically on per-hop bandwidth estimation in a scalable and user-friendly way. This observation motivates the need for a tool that combines the depth of tools like `pchar` and `mtr` with the usability of platforms like Obkio or Zabbix. Additionally, analyzing existing tools helped us identify the various metrics and parameters they employ—such as round-trip time (RTT), bandwidth, jitter, packet loss, and hop count. This provided valuable insight into which metrics are most relevant to our problem statement, guiding our efforts to incorporate the most impactful ones into our proposed tool.

III. Methodology

A. ICMP-Based Hop-by-Hop Delay and Bandwidth Estimation

The first component of our tool is built using inspiration from a traditional ICMP-based traceroute mechanism, enhanced to estimate hop-wise bandwidth using the packet-pair technique.

1) Packet Generation and Transmission:
The tool creates raw ICMP echo request (type 8) packets with a custom payload and varying TTL (Time-To-Live) values to discover each hop along the path. A raw socket is used to send these packets, which requires root privileges. For each hop, we send three probes with a TTL starting from 1 and incremented until the destination is reached or the maximum hop limit (which is set to 30 by default) is exceeded. The RTT is computed as the duration between sending the packet and receiving an ICMP Time Exceeded or Echo Reply response.

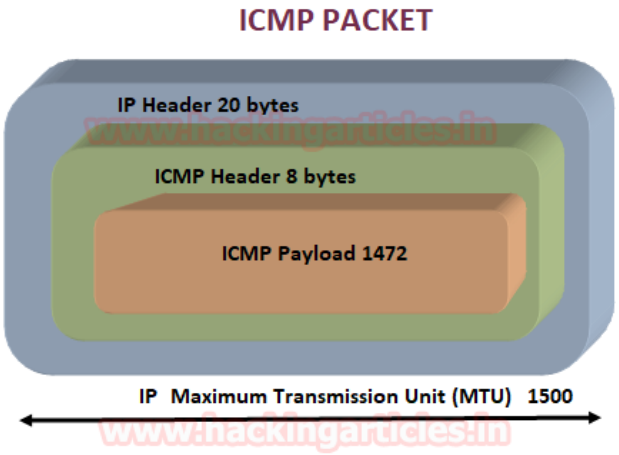

Fig. 1: ICMP Packet

2) Bandwidth Estimation and Jitter Calculation:
To estimate bandwidth at each hop, we first send a probe to determine the IP address corresponding to a given TTL and measure the round-trip time (RTT) for that hop. Once the traceroute is completed, we send 10 pairs of large ICMP packets (1400 bytes each) back-to-back to that address. The inter-arrival time between the two packets in each pair is measured using high-resolution clocks. Bandwidth is then estimated using the following formula:

$$\text{Bandwidth (Mbps)} = \frac{\text{Packet Size (bits)}}{\text{Inter-Arrival Time}(\mu s)} \quad (1)$$

This procedure is repeated for all three probes per hop, and the median bandwidth value across the 10 pairs is taken to reduce the impact of outliers and improve measurement stability.

Jitter refers to the variability in inter-arrival times of successive packets, which can be indicative of network instability. It is computed as the standard deviation of the inter-arrival intervals, capturing how much these delays deviate from their mean.:

$$\text{Jitter} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(T_i - \bar{T})^2} \tag{2}$$

- T_i: RTT (Round-Trip Time) of the i-th packet.
- $\bar{T}$: Mean RTT, the average of all RTT samples.
- n: Total number of RTT samples.
- $\sum_{i=1}^{n}(T_i - \bar{T})^2$: Sum of all squared differences between RTTs and the mean RTT.
- $n-1$: Degrees of freedom (Bessel's correction), used to avoid underestimating the population variance.

This metric provides insights into consistency of packet delivery, which is critical for latency-sensitive applications.

3) Hop Information Extraction:

Each hop's details—including hop number, IP address, RTT of first probe for 3 probes, estimated bandwidth, and the number of successful probes out of 10 (for all packet pairs)—are stored in a structured format using a custom `HopProbes` struct.

In addition to storing individual probe results, the program calculates per-hop statistics such as average RTT, jitter, and mean bandwidth. It then identifies bottlenecks using the following approach: **Instantaneous Bottleneck Link:** Determined by finding the lowest observed bandwidth value across all individual probes. This criteria provides a snapshot of the worst-case link, offering more nuanced insights into network performance.

4) Output Logging:

The results of each run are logged systematically to support both real-time diagnostics with minimal delay and long-term analysis. Primary outputs are written to a structured text file, `traceroute_output.txt`, which captures detailed probe-level metrics for every hop. For each probe, it logs the hop number, hop IP address, first round-trip time (RTT), estimated bandwidth, average rtt, jitter and the number of successful bandwidth estimations (out of 10 attempts). Once the destination is reached or the probing is complete, the file includes a conclusive summary that notes whether the destination was reached and flags any unexpected or anomalous hops.

In addition to the hop-wise trace, high-level statistics are written to a separate file, `stats.txt`. This includes:

- Total number of hops to the destination.
- Average RTT across all hops.
- Average bandwidth across all hops.
- Instantaneous bottleneck link.
- Bottleneck (Effective Bandwidth) based on minimum bandwidth per hop.

To facilitate long-term analysis and visualization, the tool also saves results in a structured CSV file named `results.csv`. Each row in this file contains the following information:

- Usage Count – the number of times the tool has been used to probe the given destination IP.
- Destination IP Address.
- Resolved Hostname.
- Timestamp of the probe run.
- Hop Number.
- Hop IP Address.
- RTT (in milliseconds).
- Bandwidth (in Mbps) for the hop.

The usage count allows the tool to distinguish multiple runs for the same destination over time, helping to track evolving performance trends. This CSV log serves as the primary data source for the visualization module, which plots RTT and bandwidth against parameters such as destination hostname, usage instance, and time of day, enabling users to monitor network health and detect bottlenecks more intuitively.

B. UDP-Based Hop-by-Hop Delay and Bandwidth Estimation

To overcome limitations of ICMP-based probing—particularly rate limiting and deprioritization by some routers—we implemented an alternative approach using UDP packets. This method is almost same as that of the ICMP method stated above, except the packets sent are UDP packets.

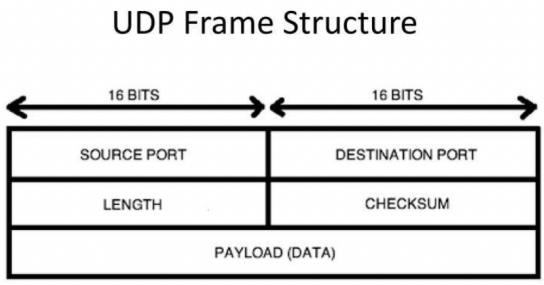

Fig. 2: UDP Packet

C. Visualization Pipeline and Bottleneck Highlighting

To intuitively present the collected network metrics, we developed a Python-based visualization pipeline using the `Pyvis` libraries. Before visualization, the raw traceroute output undergoes a parsing stage to extract structured data.

A custom Python parser is implemented to handle this task. It reads the traceroute output line-by-line and utilizes regular expressions to extract key metrics such as hop number, IP address, RTT, bandwidth, and probe success rates. The output files follow a consistent and structured format, which simplifies automated parsing and downstream processing. Special care is taken to handle exceptions, such as skipping timeout entries (represented by "*") and ignoring lines unrelated to hop data (e.g., headers or summary statistics).

Once parsed, this clean and structured data is passed to the visualization module. The pipeline then transforms the hop-by-hop metrics into an interactive, browser-rendered network graph. Each node in the graph represents a hop, and the edges between them are annotated with metrics such as RTT and bandwidth, visually highlighting performance variations along the path.

1) Graph Construction:
The traceroute path is modeled as a directed graph where:

- Each node represents a network hop, labeled by its IP address.
- Edges represent logical links between consecutive hops.
- The source node is marked as "Source," while the destination node is inferred from the last responding IP.
- The graph will have a main route and some unexpected open ended routes
- When three probes are sent per hop and different IPs are reached within a single hop, we choose the primary path based on certain criteria. If two out of the three probes return the same IP, that IP is selected as the main path. If no majority is found, the IP with the maximum number of successful probes is chosen. If there is still a tie, the IP with the minimum Round-Trip Time (RTT) is selected. Once the primary IP for the hop is chosen, it is connected to the corresponding chosen IP from the next hop. All other unexpected IPs are treated as branches originating from the chosen node of the current hop but are not connected to any further nodes in the path.

2) Attribute Encoding:
enhance visualization and enable quick diagnosis:

- Node colors encode bandwidth:
  - Green for high bandwidth
  - Orange for medium bandwidth
  - Red for low bandwidth
  - Gray for timed out requests (blocking packets)
  - Blue for source and destination
- The bottleneck link (i.e., the hop with the lowest bandwidth) is highlighted with a distinct white node node to draw immediate attention.
- Node tooltips display detailed statistics: IP address, RTT, jitter and bandwidth.

3) Interactive Output:
The final graph is exported as a self-contained HTML file using Pyvis, allowing users to explore the route interactively in any modern web browser. The visualization supports features such as zoom, pan and node hover details.

4) Network Statistics Panel: At the bottom of the HTML output, a statistics panel is injected, summarizing:

- Number of Hops needed to reach destination
- Average RTT
- Average Bandwidth
- Bottleneck Bandwidth (effective badwidth)
- Jitter

5) Time-Series Analysis Using CSV:
In addition to the hop-by-hop network graph, we generate time-series plots for RTT and bandwidth using data stored in the structured CSV file. A dedicated Python script reads this CSV file and groups the data by timestamp and hop. For each usage instance, it calculates the average RTT and bandwidth per hop. The user can interactively select a specific time window (e.g., morning, afternoon, evening) to visualize how RTT and bandwidth values fluctuate over time for a particular destination or hostname.
This feature allows users to identify temporal patterns in network conditions, such as recurring congestion during peak hours or consistently low bandwidth during specific times. Such insights are valuable for scheduling bandwidth-heavy tasks during off-peak hours or diagnosing intermittent issues that only occur at certain times.

The resulting graphs, generated using Matplotlib and Seaborn, offer a clear visual understanding of RTT and bandwidth variations over time, providing both historical context and actionable insights.

D. System Integration Architecture

The architecture of the tool is split into two key components: the low-level probing engine written in C++ and the high-level visualization module written in Python. These two components are loosely coupled via file-based I/O, which ensures modularity and ease of testing.

1) Data Flow:
The C++ backend is responsible for performing traceroute probing using either ICMP or UDP, calculating RTT and bandwidth for each hop, and writing results to `traceroute_output.txt` and `stats.txt`. These files acts as a bridge between the two layers of the tool. Once the output is generated, the Python visualizer parses this file using regular expressions and builds a graph object using the `PyVis` library. This separation allows the tool to be extended with alternative probing strategies (e.g., TCP-based) without modifying the visualization pipeline.

2) File Format and Interpretation of low level output:
The file format is kept intentionally simple and human-readable. The parser:

- Skips non-data lines such as headers and separators
- Validates the format of each row to ensure correct hop number, IP, RTT, bandwidth and jitter
- Handles timeouts and missing data gracefully

IV. Traceroute Implementation

A. Core Components and Data Structures

Our tracerouting implementation utilizes several key data structures:

- **ICMPPacket**: A structure that combines an ICMP header with a data payload, enabling customized packet creation.
- **ProbeReply**: Stores information about each probe response including IP address, round-trip time (RTT), bandwidth estimation, and bottleneck status.
- **HopProbes**: Aggregates multiple probe replies for a single hop (TTL value).
- **UnexpectedHop**: Tracks instances where replies came from unexpected IP addresses, useful for detecting network anomalies.

B. Key Constants and Their Rationale

- **MAX_PACKET_SIZE = 1472**: Defines the maximum size for ICMP packets, providing sufficient space for headers and payload. while also staying less than the MTU after which packets are fragmented.
- **DEFAULT_MAX_HOPS = 30**: Matches the standard TTL limit in most operating systems' traceroute implementations, as most Internet paths don't exceed 30 hops.
- **DEFAULT_TIMEOUT = 3**: Sets a 3-second timeout for receiving responses, balancing between waiting long enough for slow links while not delaying the overall trace excessively.
- **DEFAULT_PROBES = 3**: Uses 3 probes per hop, which is the standard in most traceroute implementations, providing sufficient data for statistical analysis without excessive traffic.
- **UDP port:** 33434 Higher numbered port which might not be in use (similar to UDP-Traceroute)

C. Main Functions and Their Implementation Details

1) Checksum Calculation:

The `calculate_checksum()` function implements the Internet Checksum algorithm (RFC 1071) for ICMP packets:

- Processes the data in 16-bit chunks, summing them up
- Handles odd-length data by padding with zeros
- Performs one's complement of the sum to generate the final checksum
- Essential for ICMP packets to be recognized as valid by network devices

2) Socket Creation:

The `create_icmp_socket()` function creates a raw socket for sending ICMP packets:

- Uses `SOCK_RAW` socket type with `IPPROTO_ICMP` protocol
- Requires root/administrator privileges
- Raw sockets are necessary because standard sockets do not allow custom ICMP packet creation

3) ICMP Packet Creation:

The `create_icmp_packet()` function builds an ICMP Echo Request packet:

- Sets type to `ICMP_ECHO` (8) and code to 0, the standard values for ping/traceroute
- Uses the process ID as part of the identifier to distinguish this program's packets from others
- Fills the data portion with 'x' characters to reach the desired packet size
- Calculates and sets the checksum to ensure packet validity

4) Note on UDP Usage:

In addition to ICMP, this tool can also leverage UDP for probing intermediate hops and estimating hop-by-hop delay and bandwidth. Unlike ICMP, UDP sockets can be created using standard `SOCK_DGRAM` sockets without requiring raw socket privileges. The tool sends UDP packets to high-numbered destination ports which are expected to be closed. Routers along the path respond with ICMP `Time Exceeded` messages when the packet's TTL expires, allowing identification of intermediate hops.

The TTL (Time-To-Live) field is set using the `setsockopt()` system call before sending each UDP packet. By incrementing the TTL across successive packets, the tool simulates traceroute-like behavior. The round-trip time (RTT) to each hop is estimated by timestamping the send and receive events. For bandwidth estimation, back-to-back UDP probe packets are transmitted, and the variation in response times helps infer dispersion patterns and potential bottlenecks. Since UDP does not require manual checksum calculation in this context, the kernel handles transport-layer encapsulation and validation.

5) Bandwidth Estimation Implementation:

The `estimate_bandwidth()` function performs bandwidth and jitter estimation as discussed earlier in the Methodology section using the following steps:

- Initializes UDP/ ICMP sockets, and sets TTL and receive timeout.
- Sends 10 pairs of back-to-back UDP packets to the destination. If the destination is set to the intermediate IP address for that hop, the response is not received.
- Measures the time gap between responses to compute inter-arrival intervals.
- Verifies that replies originate from the expected hop IP address.
- Logs unexpected replies for diagnostic purposes.
- Calculates bandwidth per pair and applies median or average filtering.
- Computes jitter as the standard deviation of the 10 RTTs.
- Returns estimated bandwidth (in Mbps), number of valid probes, and jitter (in ms).

6) Main Traceroute Function:

The `traceroute()` function performs the traceroute operation:

- Resolves the destination hostname to an IP address
- Iteratively sends probes with increasing TTL values
- For each TTL, sends 3 probes to gather statistical data

- Collects hop's IP address data and performs bandwidth estimation, RTT and jitter measurement
- Identifies potential bottlenecks
- Terminates when the destination is reached or max hops is exceeded
- Creates a new socket for each probe to avoid cross-probe interference
- Uses `inet_ntop()` with separate buffers to avoid string overwriting issues
- Sets socket timeout using `setsockopt()` to prevent indefinite waiting
- Calculates jitter (RTT variation) as a network quality indicator

7) Unexpected Hops Detection:

The `print_unexpected_hops()` function prints details of any unexpected hops encountered during the traceroute. An "unexpected hop" is one where the actual IP address differs from the planned IP address. The function outputs a formatted list to the provided output stream. If no unexpected hops are detected, it outputs an appropriate message.

8) Network Statistics Calculation:

The `calculate_network_stats()` function aggregates statistics across all hops:

- Calculates average RTT and jitter across all successful probes
- Determines average bandwidth and identifies the bottleneck (minimum bandwidth)
- Provides a summary of the overall network path characteristics

V. Visualization Tool Implementation

The visualization tool provides an interactive interface to explore network traceroute data and performance metrics. It is composed of a Streamlit-based frontend and a FastAPI-based backend, which together orchestrate data collection, processing, and display. The following subsections describe each component and the supporting modules.

A. Frontend: Streamlit Application

The frontend, implemented in `app.py`, serves as the user interface for the tool. Key features include:

- **User Inputs:** Users can enter a destination address (IP or domain name) and select the packet type for probing (ICMP or UDP) via radio buttons.
- **Execution Trigger:** On clicking the "Run" button, the frontend sends a POST request to the FastAPI backend with the chosen parameters.
- **Result Display:** After traceroute execution, the frontend displays:
  - An interactive network path map (embedded via an iframe).
  - A network topology graph.
  - The raw traceroute output and network statistics.
  - RTT and bandwidth plots filtered by time of day.

B. Backend: FastAPI Server

The backend, implemented in `main.py`, is responsible for running the traceroute utility and processing its output:

- **Command Execution:** Depending on the selected packet type, the backend runs the appropriate traceroute executable (`traceroute_icmp` or `traceroute_udp`) with root-level privileges.
- **Data Processing:** The backend reads the generated output files (e.g., `traceroute_output.txt` and `stats.txt`) and invokes supporting modules to:
  - Parse the traceroute file and generate an interactive map using `map.py`.
  - Build a network topology graph using `network.py`.
  - Process CSV data and produce performance plots (RTT and bandwidth) via `network_analysis.py`.
- **API Endpoints:** The FastAPI server provides endpoints to:
  - Run the traceroute (`/run_traceroute`).
  - Serve the interactive map (`/map`) and network topology graph (`/network_topology`).
  - Return plot images via the `/plots` endpoint.

C. Supporting Modules

Several Python modules support the backend processing and visualization:

- **`map.py`:** Uses Folium to generate an interactive map by obtaining geolocation data for traceroute IP addresses (via the ipinfo.io API) and drawing paths between successive hops.
- **`network.py`:** Employs the Pyvis library to build a network topology graph from traceroute data. It selects key nodes based on RTT and bandwidth metrics and creates a directed graph with color-coded nodes.
- **`network_analysis.py`:** Processes CSV files containing traceroute metrics using Pandas and Seaborn. It generates RTT and bandwidth plots for various times of day, providing insight into network performance.

D. Integration and Workflow

When a user submits a traceroute request:

1) The Streamlit frontend sends a POST request to the FastAPI backend with the destination and packet type.
2) The backend executes the appropriate traceroute binary with elevated privileges and collects the output.
3) The output files are parsed by the supporting modules to generate visualizations: an interactive map, a network topology graph, and performance plots.
4) The FastAPI server returns JSON data with URLs for the generated artifacts and raw output.
5) The Streamlit frontend embeds these visualizations and displays them, allowing users to interactively explore the network path and performance metrics.

This integrated visualization tool enables real-time network analysis by combining robust backend processing with an intuitive and interactive frontend.

VI. Results and Observations

To evaluate the accuracy and effectiveness of our tool, we conducted experiments targeting several public and regional endpoints using both ICMP and UDP-based probing modes. Our goal was to observe RTT behavior, estimate bandwidth across hops, find jitter to know more about network stability and reliability of results, and visualize the resulting paths to detect bottlenecks.

A. Setup and Configuration

The experiments were conducted on a Linux machine using root privileges to allow raw socket operations. All visualizations were generated using the Python-based frontend and rendered as standalone HTML files using Pyvis.

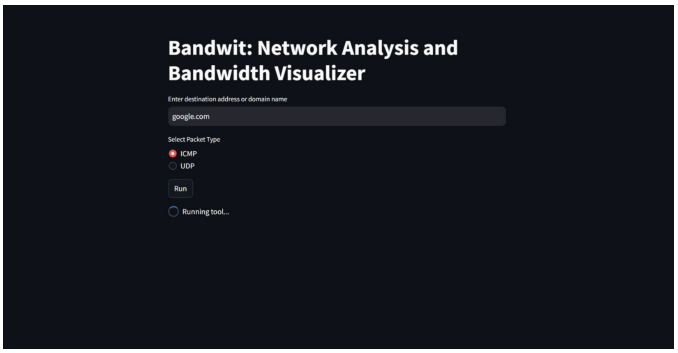

Fig. 3: BANDWIT interface

B. ICMP-Based Traceroute Results

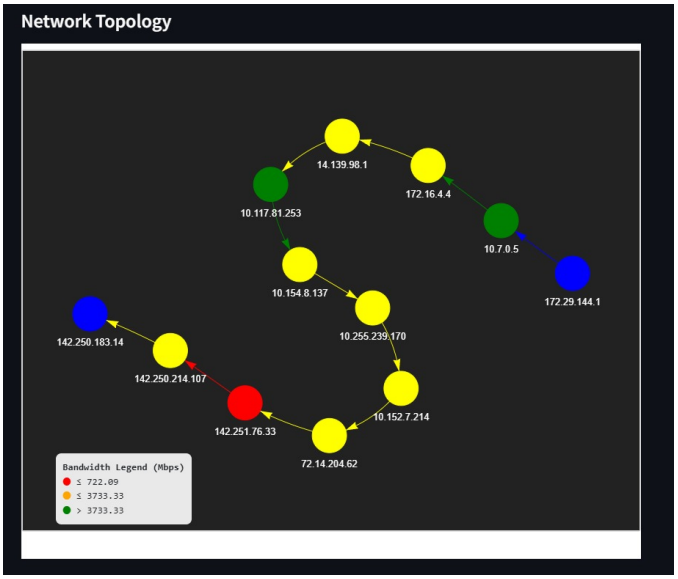

Fig. 4: Network Topology for google.com

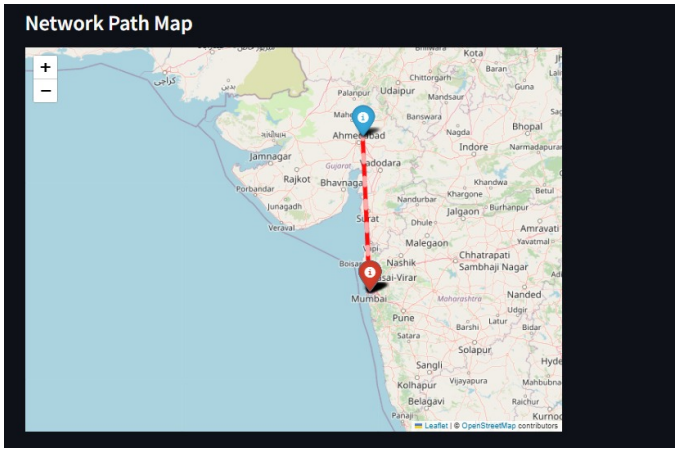

Fig. 5: Map for google.com

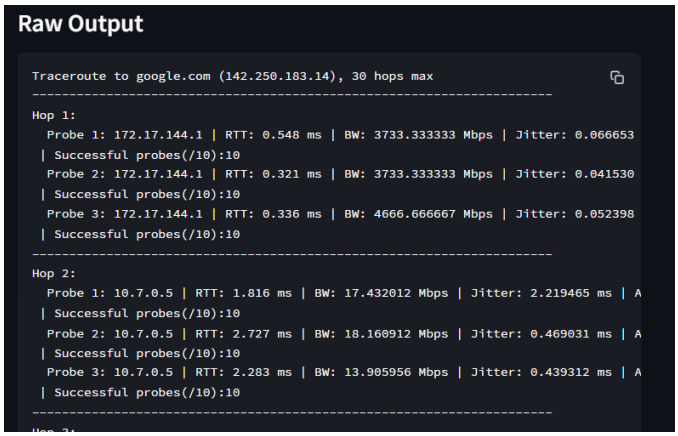

Fig. 6: Raw Output

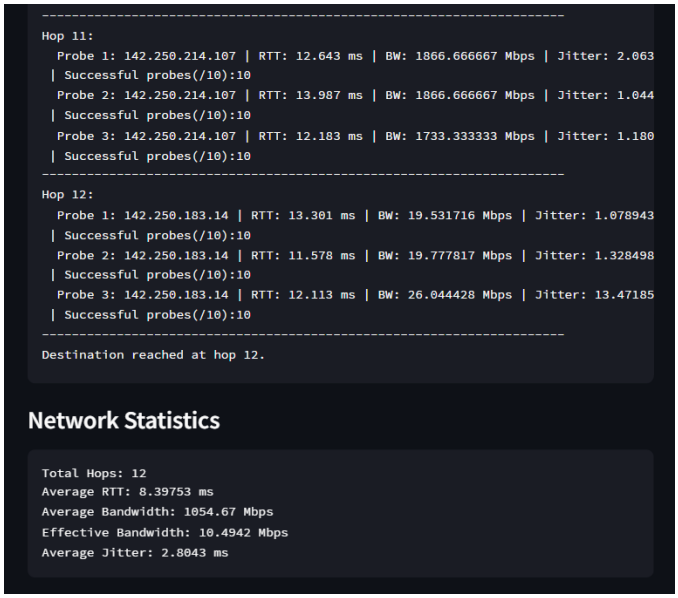

Fig. 7: Network Statistics

Figure 4 shows the visualization generated using ICMP probing toward a public DNS server (e.g., google.com). RTTs increased gradually across hops except for some outliers.

C. UDP-Based Traceroute Results

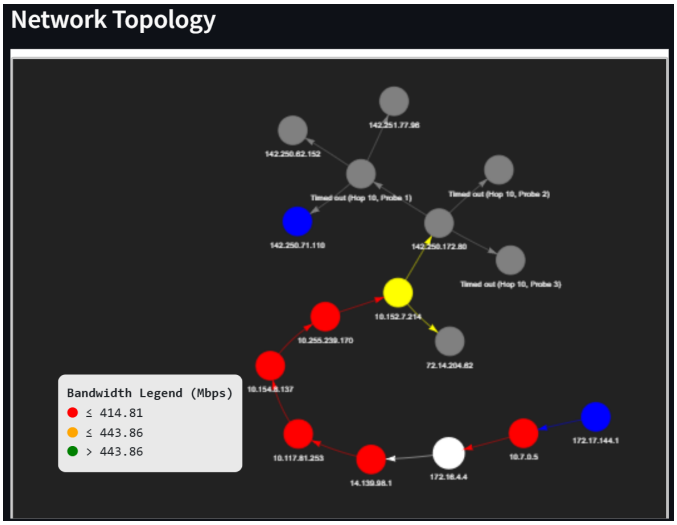

Fig. 8: Network Topology for google.com

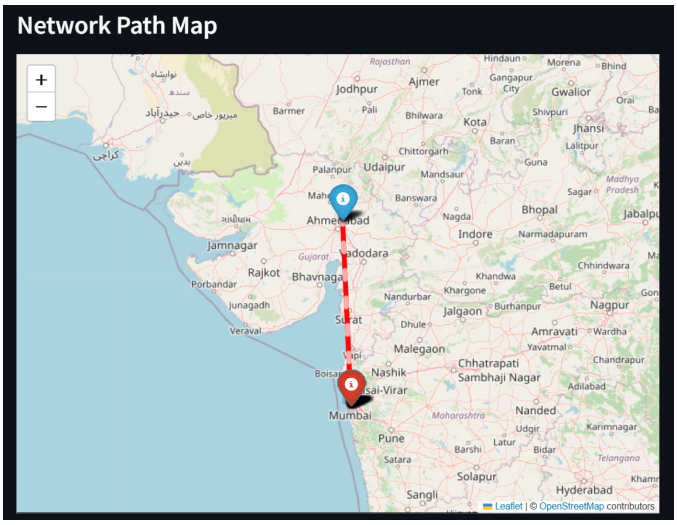

Fig. 9: Map for google.com

Figure 13 illustrates the result of a UDP-based probe to the same destination. Raw output similar to ICMP based implementation's raw output were generated. For clarity and concise report they were not added.

D. UDP-Based Traceroute vs ICMP-based Traceroute

The comparison of traceroute outputs using UDP and ICMP packets reveals distinct behaviors due to how routers and firewalls handle different protocols.

1) Destination - ims.iitgn.ac.in:

- **ICMP-based Traceroute:**

```
Traceroute to ims.iitgn.ac.in (10.0.137.79), 30 hops max
------------------------------------------------------------------
Hop 1:
  Probe 1: 172.17.144.1 | RTT: 0.739 ms | BW: 3733.333333 Mbps | Successful probes
  Probe 2: 172.17.144.1 | RTT: 0.467 ms | BW: 3266.666667 Mbps | Successful probes
  Probe 3: 172.17.144.1 | RTT: 0.418 ms | BW: 3266.666667 Mbps | Successful probes
  [Stats] Avg RTT: 0.541333 ms | Jitter: 0.141196 ms | Avg BW: 3422.222222 Mbps
------------------------------------------------------------------
Hop 2:
  Probe 1: 10.7.0.5 | RTT: 2.182 ms | BW: 42.828823 Mbps | Successful probes(/10):
  Probe 2: 10.7.0.5 | RTT: 2.158 ms | BW: 1140.973783 Mbps | Successful probes(/10
  Probe 3: 10.7.0.5 | RTT: 2.075 ms | BW: 38.219848 Mbps | Successful probes(/10):
  [Stats] Avg RTT: 2.13833 ms | Jitter: 0.0458427 ms | Avg BW: 407.340818 Mbps
------------------------------------------------------------------
Hop 3:
  Probe 1: 10.0.137.79 | RTT: 1.914 ms | BW: N/A | Successful probes(/10):0
  [Stats] Avg RTT: 1.914 ms |No unexpected hops detected.
Results saved to traceroute_output.txt, traceroute_icmp.csv, and stats.txt
nation reached at hop 3.
No unexpected hops detected.
```

Fig. 10: ICMP-based traceroute for ims.iitgn.ac.in

- **UDP-based Traceroute:**

```
Hop 1:
  Probe 1: 172.17.144.1 | RTT: 0.802 ms | BW: 3733.333333 Mbps | Successful probes
  Probe 2: 172.17.144.1 | RTT: 0.687 ms | BW: 2800.000000 Mbps | Successful probes
  Probe 3: 172.17.144.1 | RTT: 0.719 ms | BW: 3733.333333 Mbps | Successful probes
  [Stats] Avg RTT: 0.736 ms | Jitter: 0.048463 ms | Avg BW: 3422.222222 Mbps
------------------------------------------------------------------
Hop 2:
  Probe 1: 10.7.0.5 | RTT: 2.454 ms | BW: 2240.000000 Mbps | Successful probes(/10
  Probe 2: 10.7.0.5 | RTT: 2.566 ms | BW: 2520.000000 Mbps | Successful probes(/10
  Probe 3: 10.7.0.5 | RTT: 2.14 ms | BW: 13.256294 Mbps | Successful probes(/10):1
  [Stats] Avg RTT: 2.38667 ms | Jitter: 0.180313 ms | Avg BW: 1591.085431 Mbps
------------------------------------------------------------------
Hop 3:
  Probe 1: 10.0.137.79 | RTT: 1.788 ms | BW: 2240.000000 Mbps | Successful probes(
  [Stats] Avg RTT: 1.788 ms | Jitter: 0 ms | Avg BW: 2240.000000 Mbps
------------------------------------------------------------------
Destination reached at No unexpected hops detected.
Results saved to traceroute_output.txt, traceroute_udp.csv, and stats.txt
```

Fig. 11: UDP-based traceroute for ims.iitgn.ac.in

- **Bandwidth and Jitter:** In our measurements, the ICMP-based traceroute was unable to estimate per-hop bandwidth effectively. Although the initial probe packet reached the destination server (`ims.iitgn.ac.in`) and received a response, the subsequent packets used for bandwidth estimation—typically sent as packet pairs—were blocked. This was likely due to ICMP rate limiting enforced by intermediate routers or the destination itself, preventing reliable replies and thus making bandwidth calculation infeasible. In contrast, the UDP-based traceroute successfully estimated bandwidth at each hop. Since UDP packets are less likely to be rate-limited in the same manner and often trigger ICMP `port unreachable` responses from routers or endpoints, the tool was able to capture the necessary timing information for packet pairs, enabling effective hop-by-hop bandwidth and jitter analysis.

2) Destination - aws.amazon.com:

- **ICMP-based Traceroute:** The output shows multiple hop timeouts beyond Hop 5. This is likely because

many routers and intermediary devices deprioritize or outright drop ICMP Echo Requests for security or rate-limiting reasons. Additionally, the bandwidth readings drop significantly at Hop 4 and Hop 5, both marked as bottlenecks. This suggests network congestion or policy-based throttling for ICMP packets.

```
lavanya04@LavanyaPC:/mnt/c/users/lavanya/documents/github/network-analyser$ sudo ./traceroute aws.amazon.com
Traceroute to aws.amazon.com (108.158.232.78), 30 hops max
------------------------------------------------------------------
Hop 1:
  Probe 1: 172.17.144.1 | RTT: 1.219 ms | BW: 2240.000000 Mbps | Successful probes(/10):10
  Probe 2: 172.17.144.1 | RTT: 0.947 ms | BW: 2800.000000 Mbps | Successful probes(/10):10
  Probe 3: 172.17.144.1 | RTT: 0.376 ms | BW: 3266.666667 Mbps | Successful probes(/10):10
  [Stats] Avg RTT: 0.847333 ms | Jitter: 0.351295 ms | Avg BW: 2768.888889 Mbps
------------------------------------------------------------------
Hop 2:
  Probe 1: 10.7.0.5 | RTT: 5.401 ms | BW: 17.015495 Mbps | Successful probes(/10):10
  Probe 2: 10.7.0.5 | RTT: 3.858 ms | BW: 15.811277 Mbps | Successful probes(/10):10
  Probe 3: 10.7.0.5 | RTT: 4.66 ms | BW: 18.947137 Mbps | Successful probes(/10):10
  [Stats] Avg RTT: 4.63967 ms | Jitter: 0.630091 ms | Avg BW: 17.257970 Mbps
------------------------------------------------------------------
Hop 3:
  Probe 1: 172.16.4.4 | RTT: 3.217 ms | BW: 17.127535 Mbps | Successful probes(/10):2
  Probe 2: 172.16.4.4 | RTT: 4.401 ms | BW: N/A | Successful probes(/10):0
  Probe 3: 172.16.4.4 | RTT: 17.763 ms | BW: 2240.000000 Mbps | Successful probes(/10):1
  [Stats] Avg RTT: 8.46033 ms | Jitter: 6.59571 ms | Avg BW: 1128.563767 Mbps
------------------------------------------------------------------
Hop 4:
  Probe 1: 14.139.98.1 | RTT: 18.152 ms | BW: 2.446799 Mbps | Successful probes(/10):10  [!] Bottleneck
  Probe 2: 14.139.98.1 | RTT: 21.362 ms | BW: 1600.000000 Mbps | Successful probes(/10):10
  Probe 3: 14.139.98.1 | RTT: 8.755 ms | BW: 12.778099 Mbps | Successful probes(/10):10
  [Stats] Avg RTT: 16.0897 ms | Jitter: 5.34939 ms | Avg BW: 538.408299 Mbps
------------------------------------------------------------------
Hop 5:
  Probe 1: 10.117.81.253 | RTT: 6.202 ms | BW: 9.613612 Mbps | Successful probes(/10):10  [!] Bottleneck
  Probe 2: 10.117.81.253 | RTT: 2.032 ms | BW: 19.785136 Mbps | Successful probes(/10):10
  Probe 3: 10.117.81.253 | RTT: 5.13 ms | BW: 12.773234 Mbps | Successful probes(/10):10
  [Stats] Avg RTT: 4.45467 ms | Jitter: 1.7681 ms | Avg BW: 14.057327 Mbps
------------------------------------------------------------------
Hop 6:
  Probe 1: * Request timed out
  Probe 2: * Request timed out
  Probe 3: * Request timed out
------------------------------------------------------------------
Hop 7:
  Probe 1: * Request timed out
  Probe 2: * Request timed out
  Probe 3: * Request timed out
------------------------------------------------------------------
Hop 8:
  Probe 1: * Request timed out
  Probe 2: * Request timed out
  Probe 3: * Request timed out
------------------------------------------------------------------
```

Fig. 12: ICMP-based traceroute

- **UDP-based Traceroute:** Unlike the ICMP trace, the UDP-based traceroute continues to show responsive hops beyond Hop 5. The bandwidth appears more stable, and even higher beyond Hop 6 and Hop 7, indicating that these packets were routed and handled with higher priority. This is typical in enterprise and cloud networks where UDP is more likely to pass through firewalls.
- **Routing Path Differences:** The end destinations differ slightly (108.158.232.78 vs 54.192.18.9), which might reflect load balancing by AWS or differences in entry points to the cloud network. Still, the structural difference in how the traceroute progresses highlights protocol-specific handling.

E. RTT and Bandwidth across the day

This shows the variation in Round Trip Time (RTT) per hop for the domain `ims.iitgn.ac.in` during the night. Each line corresponds to a different usage count, where a usage count of 8 represents measurements taken earlier in the night, while higher usage counts (e.g., 40) correspond to measurements taken later in the night. The peak RTT is observed at the second hop for lower usage counts. However, for higher usage counts—i.e., late-night measurements—the RTT is noticeably lower, indicating that packets experience less delay and travel faster during late-night hours due to reduced network congestion.

The above figure displays the measured bandwidth per hop during the night for `ims.iitgn.ac.in`. Bandwidth

```
lavanya04@LavanyaPC:/mnt/c/users/lavanya/documents/github/network-analyser$ sudo ./traceroute_udp aws.amazon.com
Traceroute to aws.amazon.com (54.192.18.9), 30 hops max
------------------------------------------------------------------
Hop 1:
  Probe 1: 172.17.144.1 | RTT: 2.971 ms | BW: 2800.000000 Mbps | Successful probes(/10): 10
  Probe 2: 172.17.144.1 | RTT: 0.669 ms | BW: 5600.000000 Mbps | Successful probes(/10): 10
  Probe 3: 172.17.144.1 | RTT: 0.671 ms | BW: 3733.333333 Mbps | Successful probes(/10): 10
  [Stats] Avg RTT: 1.437 ms | Jitter: 1.0847 ms | Avg BW: 4044.444444 Mbps
------------------------------------------------------------------
Hop 2:
  Probe 1: 10.7.0.5 | RTT: 3.89 ms | BW: 39.997640 Mbps | Successful probes(/10): 10
  Probe 2: 10.7.0.5 | RTT: 4.761 ms | BW: 11.149621 Mbps | Successful probes(/10): 10
  Probe 3: 10.7.0.5 | RTT: 6.08 ms | BW: 14.928957 Mbps | Successful probes(/10): 10
  [Stats] Avg RTT: 4.91033 ms | Jitter: 0.900278 ms | Avg BW: 22.025406 Mbps
------------------------------------------------------------------
Hop 3:
  Probe 1: 172.16.4.4 | RTT: 6.955 ms | BW: 707.320261 Mbps | Successful probes(/10): 2
  Probe 2: 172.16.4.4 | RTT: 26.542 ms | BW: N/A | Successful probes(/10): 0
  Probe 3: 172.16.4.4 | RTT: 35.539 ms | BW: 2.352447 Mbps | Successful probes(/10): 1  [!] Bottleneck
  [Stats] Avg RTT: 23.012 ms | Jitter: 11.9333 ms | Avg BW: 354.836354 Mbps
------------------------------------------------------------------
Hop 4:
  Probe 1: 14.139.98.1 | RTT: 6.837 ms | BW: 636.544473 Mbps | Successful probes(/10): 10
  Probe 2: 14.139.98.1 | RTT: 6.195 ms | BW: 22.654511 Mbps | Successful probes(/10): 10
  Probe 3: 14.139.98.1 | RTT: 6.411 ms | BW: 21.364350 Mbps | Successful probes(/10): 10
  [Stats] Avg RTT: 6.481 ms | Jitter: 0.266728 ms | Avg BW: 226.854445 Mbps
------------------------------------------------------------------
Hop 5:
  Probe 1: 10.117.81.253 | RTT: 4.264 ms | BW: 16.745210 Mbps | Successful probes(/10): 10
  Probe 2: 10.117.81.253 | RTT: 3.56 ms | BW: 21.284552 Mbps | Successful probes(/10): 10
  Probe 3: 10.117.81.253 | RTT: 5.741 ms | BW: 21.745997 Mbps | Successful probes(/10): 10
  [Stats] Avg RTT: 4.52167 ms | Jitter: 0.90884 ms | Avg BW: 19.925253 Mbps
------------------------------------------------------------------
Hop 6:
  Probe 1: 10.154.8.137 | RTT: 72.404 ms | BW: 1422.222222 Mbps | Successful probes(/10): 10
  Probe 2: 10.154.8.137 | RTT: 16.662 ms | BW: 717.891374 Mbps | Successful probes(/10): 10
  Probe 3: 10.154.8.137 | RTT: 28.457 ms | BW: 13.403065 Mbps | Successful probes(/10): 10
  [Stats] Avg RTT: 39.1743 ms | Jitter: 23.9853 ms | Avg BW: 717.838887 Mbps
------------------------------------------------------------------
Hop 7:
  Probe 1: 10.255.239.170 | RTT: 17.392 ms | BW: 16.364545 Mbps | Successful probes(/10): 10
  Probe 2: 10.255.239.170 | RTT: 13.563 ms | BW: 18.947232 Mbps | Successful probes(/10): 10
  Probe 3: 10.255.239.170 | RTT: 17.681 ms | BW: 47.538873 Mbps | Successful probes(/10): 10
  [Stats] Avg RTT: 16.212 ms | Jitter: 1.87684 ms | Avg BW: 27.616883 Mbps
------------------------------------------------------------------
Hop 8:
  Probe 1: 10.152.7.38 | RTT: 15.272 ms | BW: 634.476052 Mbps | Successful probes(/10): 10
  Probe 2: 10.152.7.38 | RTT: 14.501 ms | BW: 1400.000000 Mbps | Successful probes(/10): 9
  Probe 3: 10.152.7.38 | RTT: 14.738 ms | BW: 1500.000000 Mbps | Successful probes(/10): 10
```

Fig. 13: UDP-based traceroute

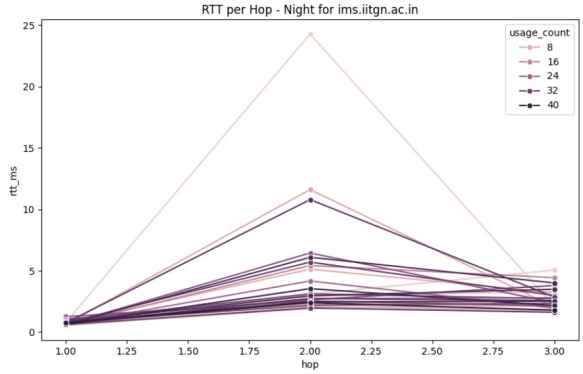

Fig. 14: rtt for ims

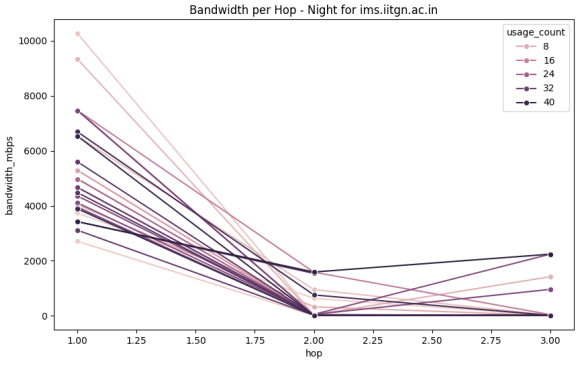

Fig. 15: bandwidth for ims

generally drops after the first hop, especially for lower usage counts. Interestingly, at higher usage counts—again, associated with late-night measurements—the bandwidth remains more stable and occasionally improves, which aligns with the lower RTT observed earlier. This indicates better link performance and fewer competing transmissions at night.

The plots presented above are specific to the **night** time interval. However, the tool also supports generation of similar RTT and bandwidth plots for other times of the day—**morning**, **afternoon**, or **evening**—allowing a comparative study of network behavior across different periods. The figure below shows the other time intervals the user can view:

Fig. 16: Other time periods

F. Multiple paths visualization

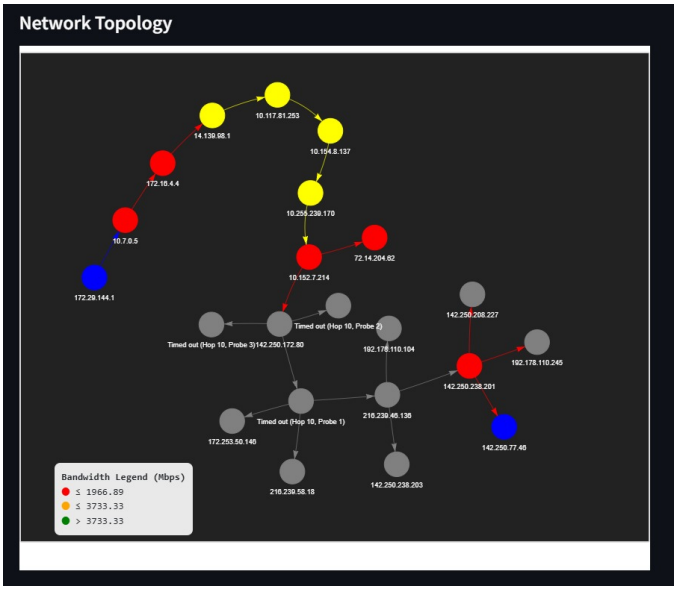

Fig. 17: Multiple Paths

VII. Comparative Analysis with Existing Tools

Our implementation extends beyond traditional traceroute tools by:

- Incorporating bandwidth estimation using packet-pair techniques
- Providing detailed statistical analysis of network performance
- Identifying potential bottlenecks based on bandwidth thresholds
- Supporting both ICMP and UDP-based probing for comprehensive path analysis

A. Comparison with MTR

We also compared our tool's outputs with that of `mtr` , a commonly used network diagnostic utility. Figure 18 and Figure 20, 21 show snapshots of the RTT and bandwidth data from both tools for the same destination.

Fig. 18: Using MTR

B. Comparison with PingPlotter

We also compared our tool's outputs with that of `PingPlotter` Figure 19 and Figure 22 show snapshots of the RTT and bandwidth data from both tools for the same destination.

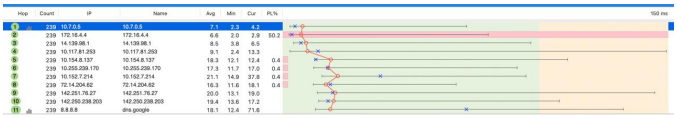

Fig. 19: Using PingPlotter

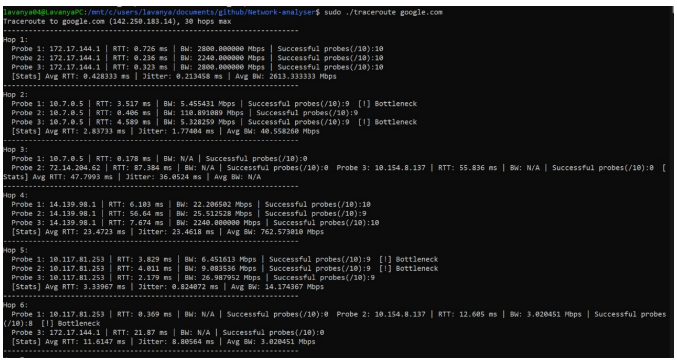

Fig. 20: Using our tool

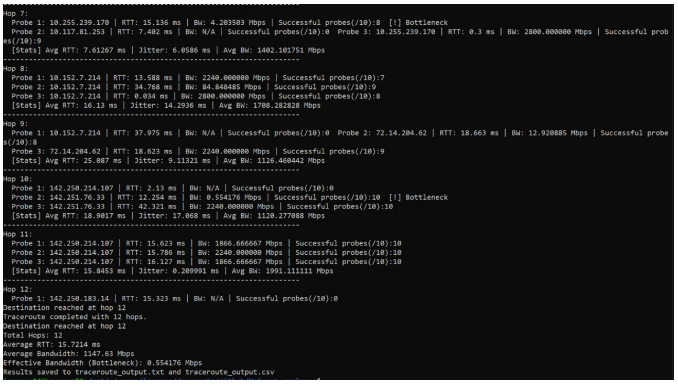

Fig. 21: Using our tool

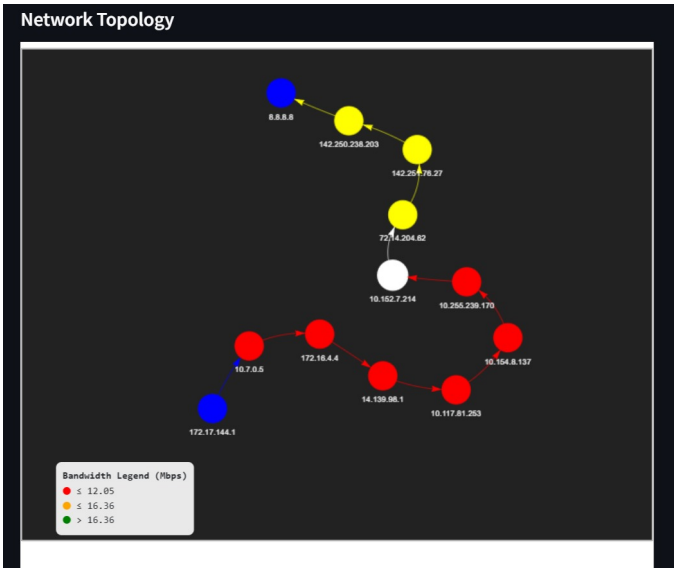

Fig. 22: Using our tool

The tool reliably traces paths to various destinations and identifies bottlenecks. RTT results are consistent with existing tools like MTR. While all tools show similar RTT progression, our tool provides additional bandwidth insights and an interactive visual representation, which `mtr` and `PingPlotter`lacks.

VIII. Challenges and Limitations

A. State-of-the-art Tools

Many existing state-of-the-art tools were either proprietary or had complex installation procedures, which hindered effective literature review and made comparative analysis more challenging.

B. Raw Socket Requirements

Both ICMP and UDP probing modes rely on raw sockets for sending and/or receiving low-level network packets. This requires elevated privileges (typically root access) on Unix-like systems, which may limit usability for non-privileged users.

C. ICMP Rate Limiting

Many routers deprioritize or rate-limit ICMP traffic, especially on intermediate hops. This causes packet loss, inflated RTTs, or missing responses, affecting the completeness and accuracy of ICMP-based traceroute results.

D. UDP Unexpected IPs

Although UDP packets are not blocked, we tend to find too many unexpected hops while using our tool, probably attributed to packets being prioritized by all routers.

E. Bandwidth Estimation Accuracy

The packet-pair dispersion technique used for estimating bandwidth is a coarse approximation. It assumes that both packets traverse the same path under similar queuing conditions. In reality, this assumption may not always hold, especially on longer or more congested paths.

F. SNMP Bandwidth Retrieval Issues

We attempted to retrieve bandwidth information using SNMP by querying the `ifSpeed` object identifier. However, the response consistently returned as `NA`. This was likely due to incorrect or restricted SNMP community string access—either the string used was not `public`, or it lacked sufficient read permissions. As a result, SNMP-based bandwidth estimation could not be performed successfully.

G. Choosing Main Path with Multiple Routes

In networks with load balancing or multipath routing, traceroute does not always return a single consistent route. Determining the ”main” path for measurement was non-trivial, and our tool had to rely on frequency and consistency of responses to infer the dominant path.

H. Geolocation Limitations

There is no open directory providing accurate geolocation coordinates for IP addresses. While some public registries like ARIN offer limited information (such as country or city), we had to rely on third-party APIs to extract latitude-longitude coordinates for plotting, which introduced dependency on external services while not being significantly better at finding the exact geolocation.

I. IPv6 Testing Constraints

We attempted to support IPv6 addresses, but were unable to fully test or validate robustness due to our internet service provider not offering a public IPv6 address. Since we were running on WSL, we also faced issues in bridging the host with it. WSL2 uses a virtualized network (via Hyper-V) and by default, it operates with NAT (Network Address Translation), which may cause problems with network connectivity between WSL and the host machine, particularly with IPv6. This limited our ability to evaluate our implementation under real IPv6 conditions. However, we validated our results for loopback address using IPv6 and it was successful.

J. Hosting Limitations

Our tool is dynamic and interactive, and thus could not be easily hosted on static hosting services like GitHub Pages. Hosting required a backend or server-side setup, making free deployment options more limited for public demonstrations.

K. Limited Cross-Platform Support

The current implementation targets Linux environments. Windows and macOS compatibility is limited due to differences in raw socket APIs and privilege management.

L. Static Destination and Timing

The current system probes a static destination and generates a static visualization snapshot. It does not yet support continuous monitoring or dynamic switching of endpoints during execution.

IX. Future Work

While our current implementation provides a functional tool for hop-by-hop delay and bandwidth estimation, there are several avenues for future improvement:

- **IPv6 Support:** We aim to fully support IPv6 in future versions. Although initial efforts were made, lack of a public IPv6 address from our ISP limited comprehensive testing and deployment.
- **Continuous Monitoring:** Currently we are logging past runs done manually by users and generate temporal comparisons to detect performance degradation or routing changes. Extending the tool for automatic periodic monitoring will allow users to track bandwidth, jitter, and route changes over time—similar to tools like SmokePing.
- **Improved Visualization and Hosting:** Due to the dynamic and interactive nature of our tool, hosting it for free (e.g., on GitHub Pages) was non-trivial. Future work could explore deployment as a lightweight web service or desktop application with better portability.
- **Modular Protocol Support:** While we currently support ICMP and UDP, future iterations could include TCP-based probing to navigate restrictive firewalls.
- **Cross-Platform Support:** Our tool is currently Linux-centric. Porting it to Windows and macOS (with OS-specific raw socket handling) will increase accessibility.
- **Integration with Existing Tools:** We can integrate tools like Wireshark to leverage their capabilities while augmenting them with our bandwidth and delay estimation features.

These enhancements would move BANDWIT closer to a production-grade, platform-agnostic network diagnostic and visualization suite.

X. Discussion and Conclusion

In this project, we developed **BANDWIT**, an end-to-end network diagnostic tool that augments traditional traceroute capabilities by incorporating both ICMP and UDP-based probing, advanced metric computation, and interactive visualization. By combining low-level network probing with high-level visual analytics, BANDWIT enables deeper inspection of network path characteristics such as latency and bandwidth.

Our experiments showed that ICMP probing is effective for basic reachability checks and round-trip time (RTT) measurement. However, it suffers from limitations in environments where ICMP packets are deprioritized or rate-limited, common in enterprise networks and firewalls that aim to mitigate DoS attacks. This can result in inconsistent hop discovery and unreliable RTT data. To address this, UDP-based probing was implemented as a fallback.

The backend of the tool, written in C++, performs the probing operations and logs detailed per-hop statistics including RTT, jitter and estimated bandwidth. These results are then parsed by a Python-based visualization module, which constructs an interactive network graph using Pyvis. This modular file-based communication approach allowed for clean separation between computation and visualization, facilitating flexibility and reusability.

Comparative analysis showed that BANDWIT's output closely matched established tools like MTR in terms of hop count and RTT. Beyond that, our tool also provides bandwidth estimates per hop—offering a richer diagnostic view. The interactive visualizations made it easy to pinpoint high-delay or low-bandwidth links, which is particularly useful for diagnosing performance bottlenecks in complex network topologies.

This ease of use, combined with the depth of insight provided, makes BANDWIT suitable not just for academic exploration, but also for practical network monitoring and diagnostics.

In conclusion, BANDWIT bridges the gap between traditional command-line network tools and modern interactive analysis. Its hybrid probing strategy, extensible design, and visual output capabilities provide actionable insights into network performance, making it a valuable tool for students, researchers, and network administrators alike.

Acknowledgment

We would like to express our sincere gratitude to Prof. Sameer G. Kulkarni for his invaluable guidance and constant encouragement throughout the course of this project. His insights and feedback were instrumental in shaping the direction of our work.

We also extend our thanks to the teaching assistant, Mr. Yasir Mohi Ud Din, for his continuous support, timely clarifications, and helpful suggestions that greatly contributed to our understanding and successful implementation of various technical aspects. The learning environment fostered by the

instructional team played a crucial role in the completion of this project.

REFERENCES

[1] J. Postel, "Internet Control Message Protocol," RFC 792, Sep. 1981. https://www.rfc-editor.org/rfc/rfc792

[2] C. Dovrolis, P. Ramanathan, D. Moore, "What Do Packet Dispersion Techniques Measure?" in IEEE INFOCOM, 2001. https://ieeexplore.ieee.org/document/916634

[3] Abdesol, "Let's Make a Trace Routing Tool from Scratch with Python," Medium. https://abdesol.medium.com/lets-make-a-trace-routing-tool-from-scratch-with-python-f2f6f78c3c55

[4] A. Downey, "Clink (Characterize Links)," allendowney.com. https://allendowney.com/research/clink

[5] J. Pszczolowski, "Traceroute Implementation in Python," GitHub. https://github.com/jpszczolowski/traceroute

[6] Itfat, "Traceroute App," GitHub. https://github.com/itfat/Traceroute-App

[7] ESNet, "iPerf," GitHub. https://github.com/esnet/iperf

[8] AdityaC4, "Geo Traceroute," GitHub. https://github.com/AdityaC4/geo_traceroute

[9] CAIDA, "Bandwidth Estimation Metrics," caida.org. https://www.caida.org/catalog/papers/2003_bwestmetrics/bwestmetrics.pdf

[10] Linux Man Pages, "traceroute(8) - Linux manual page," man7.org. https://www.man7.org/linux/man-pages/man8/traceroute.8.html

[11] Y. Mathur, "Understanding Guide: ICMP Protocol with Wireshark," HackingArticles.in. https://www.hackingarticles.in/understanding-guide-icmp-protocol-wireshark/