CSCE 3401-02
Instructor: Dr. Amr El-Kadi

OS Project Report: Network Monitor

Ramy Badras ..... **Layla Mohsen** ..... **Knzy Elmasry** ..... **Yuhan Shao**
900194248 ..... **900202391** ..... **900202766** **900237845**

Contents

1 Introduction

1.1 Motivation

In the digital age, network monitoring has become an essential aspect of maintaining and optimizing network performance. Both individuals and organizations rely heavily on their network infrastructure for various activities, ranging from casual internet browsing to critical business operations. Understanding network traffic, detecting anomalies, and managing bandwidth are crucial for ensuring smooth and efficient network usage.

1.2 Importance of Network Monitoring:

Network monitoring tools provide invaluable insights into the network's behavior, allowing administrators to identify and resolve issues promptly. These tools help in:

- Detecting Network Bottlenecks: By analyzing the traffic, administrators can identify points of congestion and take corrective actions.
- Ensuring Security: Monitoring tools can detect unusual patterns that may indicate security breaches or malicious activities. Optimizing Performance: By understanding the network load and usage patterns, resources can be allocated more effectively to improve overall performance.
- Compliance and Reporting: Many industries have regulatory requirements for network monitoring and reporting. Proper tools help in maintaining compliance.

The primary objective of this project is to develop a comprehensive network monitoring tool using Rust and Tauri. In summary, this network monitoring tool seeks to enhance the ability of users to manage and optimize their network usage effectively. Through this project, we aim to bridge the gap between complex network monitoring solutions and user-friendly, accessible tools.

2 Project Overview

2.1 Programming Language: Rust

Rust was chosen as the primary programming language for this project due to its emphasis on safety, speed, and concurrency. Rust's ownership model ensures memory safety without the need for a garbage collector, which is crucial for a network monitoring tool that needs to handle real-time data efficiently. Additionally, Rust's concurrency model allows us to build a highly responsive and multi-threaded application, capable of managing multiple tasks simultaneously without performance degradation.

2.2 Build from Scratch

This project is built entirely from scratch, without extending any existing templates or prototypes. This approach allows us to tailor the tool to meet our specific requirements and ensures that we fully understand and control every aspect of the application.

2.3 Supported Systems: Linux & MacOS

The network monitoring tool is designed to support both Linux and MacOS platforms. These operating systems are widely used in both personal and professional environments, making them ideal targets for our application. By supporting these platforms, we ensure that our tool can be used by a broad audience, including network administrators, IT professionals, and everyday users.

2.4 Major Functionalities

- Real-time Monitoring: Track network packets and display real-time data.
- otifications: Alert users when new programs join the network or when specific thresholds are exceeded.
- Data Extraction and Analysis: Extract data in various formats for in-depth analysis.
- Customized Alerts: Notify users within the app or via email about critical events and anomalies.
- Graphical User Interface: Provide an intuitive and user-friendly interface for easy interaction.

3 Early Research & Initial Design

3.1 Evaluation of Available Tools

During the early research phase of our project, we evaluated several network monitoring tools to understand their strengths and weaknesses. This helped us identify key features and functionalities to incorporate into our tool. The primary tools we assessed were Darkstat, VnStat, and ZABBIX.

3.2 Our Design

Based on our evaluation, we decided to integrate the most useful features from these tools into our network monitoring tool. Our goal was to create a comprehensive, user-friendly application that caters to both non-professional and professional users.

3.2.1 Target Users

Our network monitoring tool is designed for a wide range of users:
1. Non-professional users: Individuals looking to monitor their home network usage and performance.
2. Professional users: IT administrators and network professionals managing larger network infrastructures.

3.2.2 Choice of Platform: App

We chose to develop our tool as a standalone application using Tauri, which develops apps with Rust as backend and supports multiple languages for front end such as javascript and react and many other for the frontend. This approach ensures a high-performance, secure, and cross-platform solution that can run on both Linux and MacOS systems.

3.2.3 Flow Technology

Flow technology is essential for monitoring and analyzing network traffic. It involves collecting and analyzing flow data, which summarizes traffic characteristics such as source and destination IP addresses, port numbers, and packet counts. The common flow technologies we considered include:

- NetFlow (Cisco): Tracks all packets, suitable for detailed analysis but less scalable for high-traffic networks.
- JFlow (Juniper): Similar to NetFlow, tracks all packets but with proprietary implementations.
- sFlow (Sampled Flow): Uses packet sampling, making it more scalable for high-traffic networks but less detailed.

We decided to implement sFlow due to its scalability and ability to provide a broad overview of network traffic while maintaining performance.

Feature	Darkstat	VnStat	ZABBIX
Real-time Monitoring	Yes, web-based and command-line	No, focuses on historical data	Yes, with extensive real-time monitoring
Traffic Analysis	Yes, provides detailed traffic data	Yes, but primarily historical	Yes, comprehensive traffic analysis
Alerting and Notifications	Limited, primarily relies on user monitoring	No, not built-in	Yes, advanced alerting and notifications
Historical Reporting	Yes, via web interface	Yes, detailed historical logs	Yes, extensive historical reporting
Protocol Analysis	Limited	No	Yes, supports multiple protocols
Application Performance Monitoring (APM)	No	No	Yes, monitors networked applications
Security Monitoring	Limited	No	Yes, advanced security features
Integration	Limited	Yes, with other tools	Yes, integrates with various systems
User Interface	Web-based and command-line	Command-line only	Web-based, user-friendly interface
Scalability	Limited, small design	Good for small setups	Highly scalable for large environments
Platform Support	Linux, BSD	Linux, BSD	Linux, Windows, MacOS

Figure 1: Evaluation on Available Tools

3.2.4 Functionalities

To address the needs of our target users and incorporate the strengths of the evaluated tools, our network monitoring tool includes the following functionalities:

- Real-time Monitoring
- Traffic Analysis
- Alerting and Notifications
- Historical Reporting
- Protocol Analysis
- Application Performance Monitoring

4 Architecture

4.1 Version1: main() as controller

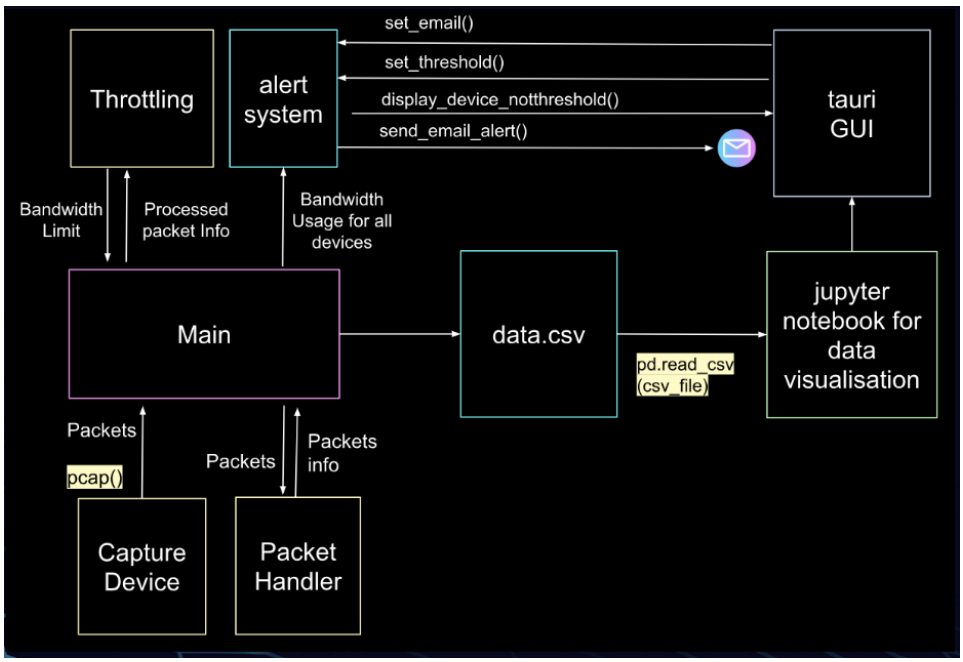


Figure 2: Old Architecture

This is not efficient and not taking advantage of multi-threading to achieve parallelism, so we banned this architecture and developed a new multi-threading one to achieve better performance and efficiency.

4.2 Version2: Multi-Threading Architecture

5 Modules and Corresponding Design Decisions

Our network monitoring tool is designed with various modules, each serving specific functionalities to achieve comprehensive network monitoring and analysis. This section details the key modules and the design decisions made during the development process. (The order follows the development timeline)

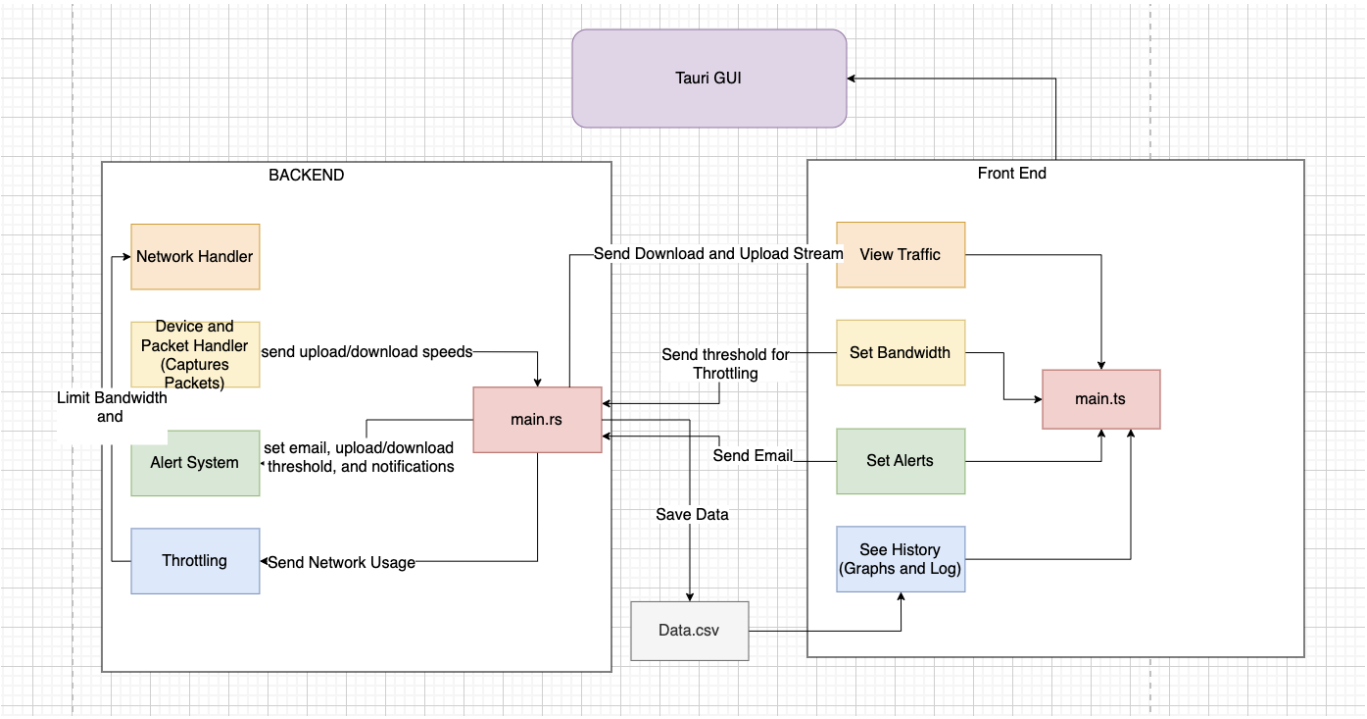


Figure 3: Multi-Threaded Architecture

5.1 Real-time Monitoring

We used multiple libraries and system-calls for efficient network traffic measurement. 'Pcap' is used at the outset to select the desired network interface and capture packets as they are transmitted. System calls, including poll() for waiting on incoming packets and mmap() for efficient capture, are employed to manage the raw socket used for capturing. Once capture is complete, close() is used to properly shut down the socket. 'pnet' then takes over to analyze the captured packets. It calculates transmission rates by summing the sizes of parsed packets within a one-second interval. The standard library is utilized to measure the time elapsed between packets, allowing for the calculation of download and upload rates which are then printed every second. This combined approach provides a comprehensive picture of network traffic, encompassing both interface selection and detailed rate analysis.

5.2 Tracking Packets Based on Processes

- Our initial investigation considered employing Pnet for process network traffic mapping. However, this library lacked essential functionality: Process Identifier (PID) support. Subsequently, we evaluated the use of lsof for the same purpose. While lsof offered PID information, it did not provide the necessary network upload/download rate data. This limitation initially presented an obstacle in definitively mapping traffic to specific processes.

- Netinfo emerged as a potential solution due to its seemingly comprehensive capabilities. Unfortunately, the lack of readily available implementation details and documentation online hampered our efforts to utilize this extension. Despite its apparent suitability, netinfo presented a further challenge in the form of dependency conflicts.

- Given these considerations, we ultimately opted to leverage lsof in conjunction with Pcap and Pnet. This combined approach allowed us to extract critical packet information, such as source and destination ports. By feeding this data into lsof, we were able to glean additional details about the associated processes, effectively achieving our traffic-to-process mapping objective.

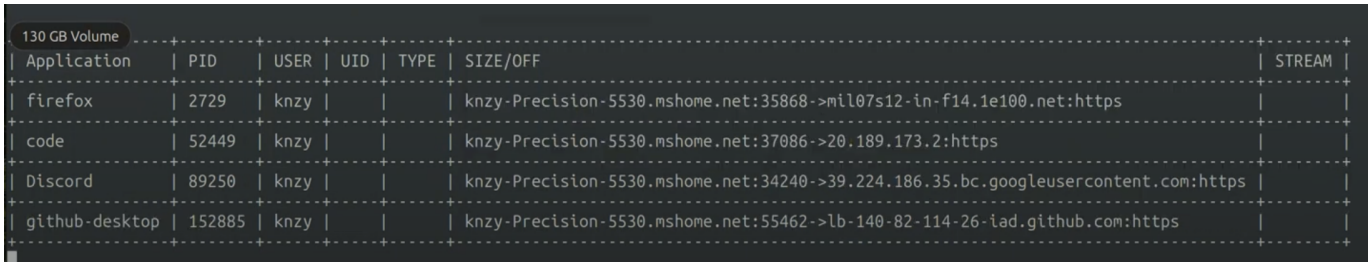


Figure 4: Tracking Processes

5.3 Notification of New Program Joined

The system continuously monitors active network connections and their associated processes. When a new process establishes a network connection, the tool generates a notification to inform the user of the new program's activity.

5.4 Historic Data Extraction

- Method 1 Buffer: Initially, we used a buffer to temporarily store captured packets before processing. This method had several drawbacks, including high memory usage and limited scalability. The buffer could overflow if not processed quickly, leading to data loss. However, the buffer method consumed a significant amount of memory, especially under high network traffic. Also, this method lacked parallelism, resulting in poor scalability and performance bottlenecks.
- Method 2: Multi-format Multi-threading Writing (Appending) To overcome the limitations of the buffer method, we adopted a multi-threaded approach. Each thread handled specific tasks, such as capturing packets, processing data, and writing to the output file in different formats.
- Reliable Data Management: Writing data in different formats (e.g., CSV, .pcap) ensured better data organization and reliability.

5.5 Email Notification

Implementation in Rust: The email alert system was implemented using the 'lettre' crate in Rust. The system monitored network activity and sent email alerts when specific thresholds were exceeded. However, there are security concerns like the authentication – there is lack of secure authentication for sending emails to prevent unauthorized access.

5.6 Real-time Graphics

For real-time data visualization, we initially used Python due to its extensive libraries and support for graphical representations. 'matplotlib' and 'pandas' were used to create various charts and graphs.We explored Rust's capabilities for generating real-time graphics but faced challenges due to limited library support. Ultimately, we overcame the issues and displayed real-time graphics in Rust.

5.7 Control C and Other Signal/Exception Handling

Signal handling was crucial for ensuring graceful termination of the program and preserving data integrity. The 'ctrlc' crate in Rust was used to handle Control C signals and other exceptions, allowing the program to perform necessary cleanup tasks before exiting.

5.8 Statistics and Graphics in Python

In Python:

- Pie Chart: Displayed data distribution by source, with a separate chart for upload and download data.
- Line Graph: Showed the trend of download and upload data over time, allowing for dynamic time filtering to analyze specific periods.
- Dynamic Time Filter: Implemented interactive time filtering using ‘matplotlib‘ and ‘seaborn‘, enabling users to adjust the time range and view data trends for selected intervals.

6 Final Tauri Project with GUI

Our final network monitoring tool was built using Tauri for the frontend and Rust for the backend. This section describes the overall architecture, specific features, frontend-backend integration, and the challenges we faced during development.

6.1 Frontend (HTML, CSS, JavaScript/TypeScript)

- index.html: The main HTML file containing the structure of the GUI.
- styles.css: The CSS file for styling the GUI components.
- main.ts: The TypeScript file handling frontend logic and interaction with the backend.
- tauri.conf.json: The configuration file for Tauri settings and permissions.

6.2 Backend (Rust)

- main.rs: The main Rust file containing the core logic for packet capturing, data processing, and communication with the frontend.
- email.rs: A module for handling email notifications using the ‘lettre‘ crate.
- packet_handler.rs: A module for handling different types of network packets (IPv4, IPv6, ARP).
- Cargo.toml: The configuration file for managing dependencies and project settings.

6.3 Data Storage

- data.csv: The CSV file where network traffic data is stored for historical analysis.

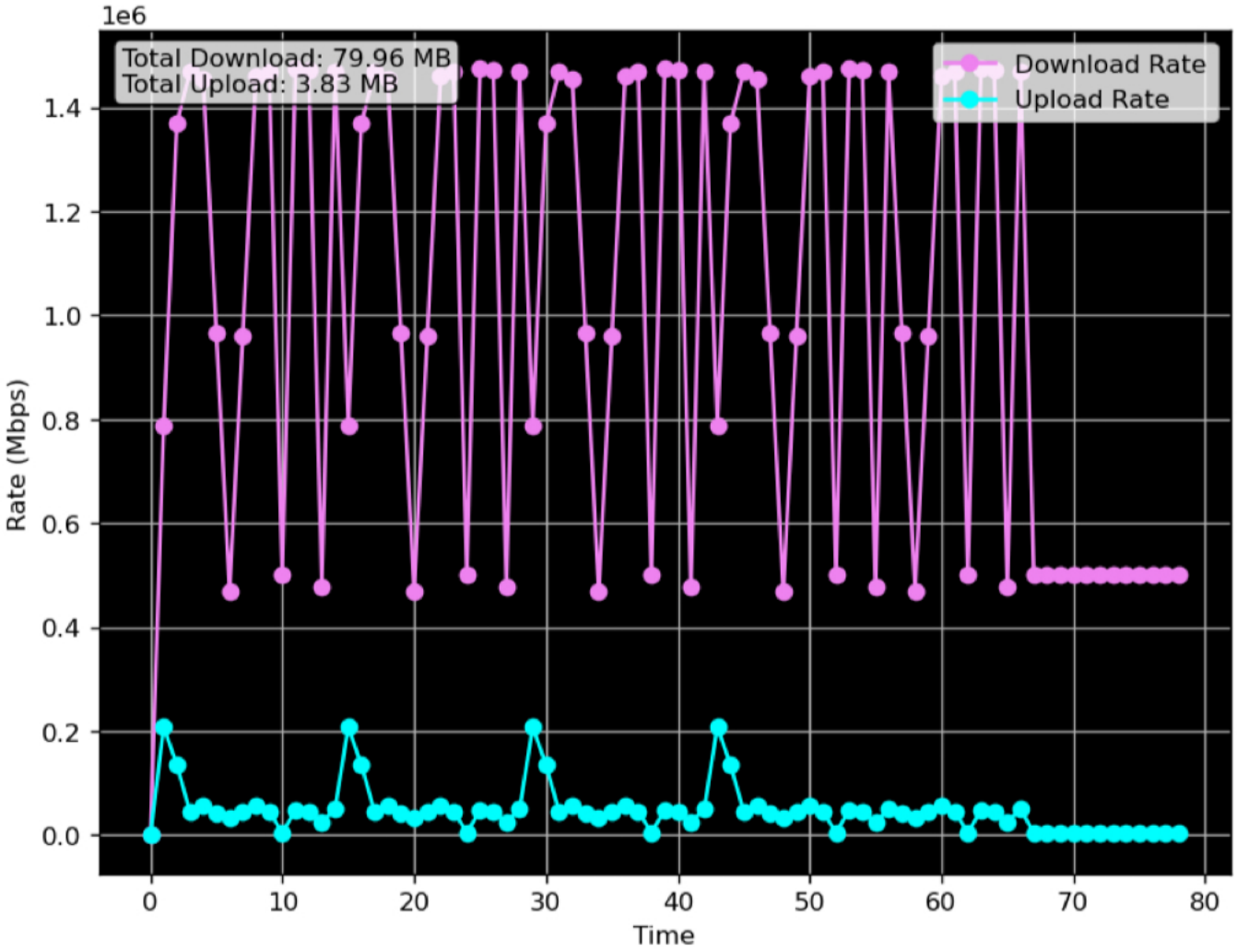
6.4 GUI Layout

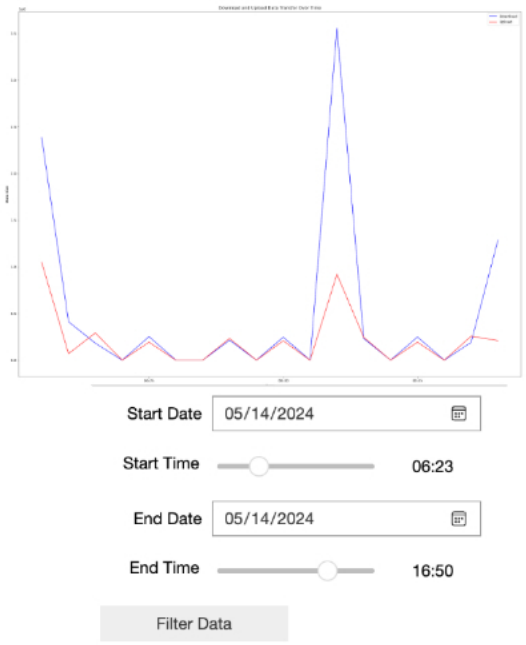
6.4.1 View Traffic (Real-time)

The real-time traffic monitoring feature allows users to see live data on network traffic. The Tauri frontend receives data from the Rust backend, which captures and processes network packets. The data is displayed using Chart.js, providing a graphical representation of upload and download rates over time.

6.4.2 Set Bandwidth

Users can set bandwidth limits for both upload and download rates through the Tauri GUI. The limits are sent to the Rust backend, which applies them using the ‘tc‘ command on Linux or PF rules on macOS. The backend continuously monitors network traffic and enforces the specified limits.





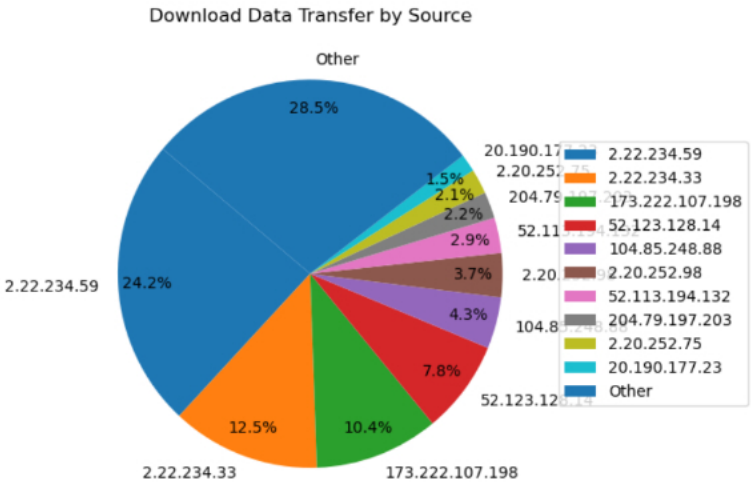


Figure 5: Real Time Graphic & Statistical Graphics in Python

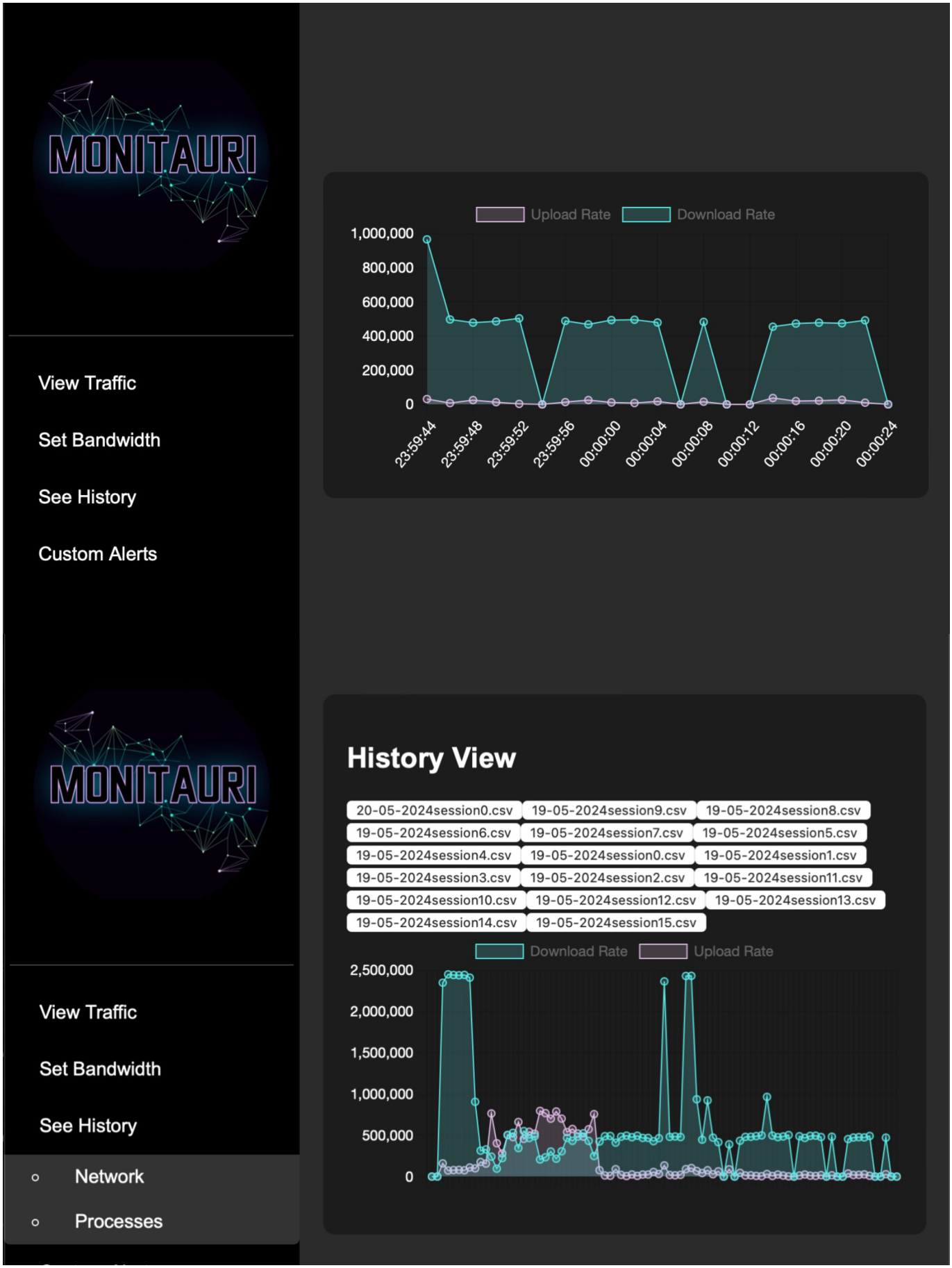


Figure 6: GUI: View Traffic & See History

6.4.3 View History

The tool allows users to view historical data on network usage. The data is stored in a CSV file and can be analyzed using a Jupyter notebook. The history view in the Tauri GUI provides an interface to select and visualize past network activity of a period of time.

6.4.4 Set Notification

Users can configure email notifications for bandwidth usage thresholds. The Tauri GUI collects the user's email address and the upload/download limits, which are sent to the Rust backend. The backend uses the 'lettre' crate to send email alerts when the specified thresholds are exceeded.

7 Evaluation & Reflection

7.1 Check-List

What is done:	**What is left:**
• Real-time network traffic monitoring with graphical display. • Bandwidth limit setting and enforcement. • Historical data storage and visualization. • Email alert system for bandwidth usage thresholds. • User-friendly Tauri GUI for interaction with the tool. • Implemented a multi-threaded approach for efficient data processing. • Used Rust for secure and high-performance backend operations.	• Individual process monitoring • Integrating additional network analysis features (e.g., protocol analysis). • Improving error handling and robustness of the backend. • Integrated Jupyter notebook for advanced data visualization.

Table 1: Project Check-list

7.2 Difficulties Encountered

- Front-back end data transmission: Ensuring efficient and seamless communication between the Tauri frontend and Rust backend was challenging. We had to handle asynchronous data updates and manage state across both ends.
- User interactivity: Providing a responsive and user-friendly interface required careful design and optimization. Handling dynamic inputs and real-time updates added complexity.
- Code integration from each member: Merging code from different team members posed integration challenges. The email setup page, for example, required coordination to ensure it worked within the overall system.

- Difficult to debug (unfamiliar with the language): Debugging in Rust and Tauri was challenging due to our initial unfamiliarity with the languages and frameworks. We faced issues with asynchronous handling and multi-threading that required deep dives into documentation and community resources.

7.3 Discussion & Reflection

Our Network Monitor project represents a significant achievement in building a comprehensive tool from scratch that offers real-time network traffic monitoring, bandwidth management, historical data analysis, and alert notifications. Using Rust for backend development ensured high performance and security, while Tauri facilitated creating a user-friendly graphical interface.

This project has been a rewarding journey, blending theoretical knowledge with practical implementation.Integrating Rust with Tauri for seamless communication between the frontend and backend posed initial challenges. However, by leveraging Tauri's API and careful design of data flow, we achieved efficient integration. It has equipped us with valuable skills in systems programming, multi-threading, and user interface design. The Network Monitor tool stands as a testament to our dedication and collaborative efforts, providing a robust solution for real-time network monitoring and management. As we move forward, we are excited to continue refining and expanding the tool's capabilities, making it even more powerful and user-friendly.