A blue and black logo

AI-generated content may be incorrect.

Data Structures And Algorithms

Assignment 2 | Network Monitor

October 25, 2025

Anas Norani

501231

Table of Contents

[1. Understanding of Data Structures: 2](#_Toc212328233)

[2. Understanding of Network Processing: 2](#_Toc212328234)

[3. Packet and Capture Management: 3](#_Toc212328235)

[4. Dissection and Filtering: 3](#_Toc212328236)

[5. Replay, Error Handling, and Demonstration: 3](#_Toc212328237)

[6. Conclusion: 4](#_Toc212328238)

[7. GitHub Repository: 4](#_Toc212328239)

[Final Evaluation Summary 5](#_Toc212328240)

# 1. Understanding of Data Structures:

In this assignment, I implemented a **Network Monitor** system using **custom stacks and queues in C++**.  
The goal was to manage network packets captured through raw sockets on a Linux system while applying my understanding of core data structures.

* **Stack Usage:**  
  I designed a custom **stack** to dissect packet layers (Ethernet → IP → TCP/UDP). Each captured packet was pushed layer-by-layer, and the stack enabled easy removal and inspection of headers using the LIFO principle. This clearly modeled how packets are encapsulated and de-encapsulated during network transmission.
* **Queue Usage:**  
  I implemented a **queue** to manage continuous packet flow. Captured packets were enqueued when received and dequeued once processed or filtered. This FIFO design allowed efficient handling of real-time data and ensured that no packet was dropped or reprocessed.

Both data structures were coded **from scratch without external libraries**, using pointers and arrays for dynamic memory handling and error checking (underflow/overflow).

# 2. Understanding of Network Processing:

I demonstrated a deep understanding of how **data structures integrate with network processing**.

* The system uses a **raw socket** to capture packets directly from the network interface.
* Every packet is parsed layer-by-layer using my stack implementation, allowing me to view Ethernet, IP, and transport-layer information.
* The **queue mechanism** reflects real-world buffering: packets enter the capture queue, are filtered, and then replayed in sequence.
* I incorporated **IP addressing, packet size calculations, and protocol identification (IPv4, IPv6, TCP, UDP)** manually, which enhanced my grasp of how packets flow through network layers.

# 3. Packet and Capture Management:

My program performs **continuous packet capture and storage** on a single network interface using root privileges.

* Each packet record includes: unique ID, timestamp, size, source IP, destination IP, and raw data.
* The capture loop runs for over one minute, continuously adding and removing packets from the custom queue.
* Memory management and buffer allocation are carefully handled to prevent overflow.

This structure ensures that the system can handle live traffic efficiently while maintaining performance stability.

# 4. Dissection and Filtering:

I implemented complete **packet dissection and filtering** logic.

* **Dissection:**  
  Each packet is processed through custom parsers for **Ethernet, IPv4, IPv6, TCP, and UDP**. These parsers extract critical information such as MAC addresses, IPs, port numbers, and payload sizes.
* **Filtering:**  
  The program allows the user to specify **source and destination IPs** for live filtering. Packets exceeding 1500 bytes are ignored after the threshold to maintain replay reliability.
* **Performance:**  
  The algorithm continuously checks packets in the queue, moving filtered ones into a replay list. The lightweight structure ensures minimal CPU load during long captures.

# 5. Replay, Error Handling, and Demonstration:

I added a **replay system** that re-sends filtered packets back into the network.

* Each replay includes a **calculated delay** (Delay = PacketSize / 1000 ms) for realistic timing.
* In case of an error, packets are retried **up to 2 times** and then moved into a **backup queue** for later replay.
* The main function demonstrates:
  + Continuous capture for 1 minute.
  + Layer-by-layer dissection.
  + Filtering between selected IPs.
  + Replay with error handling and logging.

# 6. Conclusion:

This assignment gave me a strong practical understanding of how data structures directly influence network programming.  
By implementing stacks and queues manually, I learned how to control memory, manage real-time data, and model complex system behaviour efficiently.  
The project operates smoothly under continuous load, satisfying all functional and conceptual requirements.

# 7. GitHub Repository:

This assignment gave me a strong practical understanding of how **data structures directly influence network programming**.  
By implementing stacks and queues manually, I learned how to control memory, manage real-time data, and model complex system behaviour efficiently.  
The project operates smoothly under continuous load, satisfying all functional and conceptual requirements.

**GitHub Link:** [*https://github.com/anasnorani1/DSA\_Assignment2*](https://github.com/anasnorani1/DSA_Assignment2)

**Repository Contents:**

* network\_monitor.cpp – main implementation file
* README.md – usage and setup instructions
* report.pdf – final submission report

# Final Evaluation Summary

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Criteria | Description | | Understanding of Data Structures | Comprehensive explanation of stacks and queues with real-world use | | Understanding of Network Processing | Deep, correct application of network concepts | | Packet and Capture Management | Efficient continuous capture and storage | | Dissection and Filtering | Complete protocol parsing and filtering | | Replay, Error Handling, Demonstration | Reliable replay system with 2-retry logic and demo | |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |