

**CS-250-Data Structures & Algorithms**

**ASSIGNMENT 2: Network Monitor**

**NAME: SARA FAWAD**

**CMS ID: 509615**

**Network Packet Monitor using Custom Stack and Queue**

### **Objective:**

The assignment requires to develop a network packet monitor in C++ that would enable such operations as capturing, filtering and replaying network packets through custom implementations of stacks and queues without using external data structure libraries. The project aims at understanding data structures as such, handling real time data, and parsing protocols all on a Linux platform.

### **Introduction:**

## The tools for analyzing network packets remain vital in checking and managing the traffic flowing through any given network interface. This project presents the basic development of such a tool capable of capturing, dissecting, filtering, and replaying packets via raw sockets in C++. To enable and facilitate efficient processing of network data, the project implements custom Stack and Queue data structures.

## •Stacks are used to parse and dissect layered network protocols (Ethernet, IPv4, IPv6, TCP, UDP).

## •Queues are used for storing packets and filtering and replaying them in a continuous loop.

## **The project reinforces one's understanding of:**

## • Data structures (Stacks, Queues)

## • Real-time data management

## • Algorithm design

## • Networking concepts and socket programming

## **Features:**

* Continuous packet capture (raw socket)
* Packet dissection (Ethernet, IPv4, IPv6, TCP, UDP)
* Custom Stack and Queue for layer and packet management
* IP-based packet filtering
* Packet replay with error handling (2 retries)
* Real-time packet delay estimation

### **Functional Overview:**

| **Feature** | **Description** |
| --- | --- |
| **Packet Management** | Each packet is stored with an ID, timestamp, raw buffer, source and destination IPs. Packets are added/removed continuously. |
| **Packet Capture** | Uses a raw socket (AF\_PACKET, SOCK\_RAW) to continuously capture packets on a given interface. |
| **Packet Dissection** | A **custom Stack** is used to parse layers: Ethernet → IP → TCP/UDP. Each layer is pushed and popped manually. |
| **Filtering** | Filters packets by source/destination IPs. Oversized packets (>1500 bytes) are ignored after a threshold. |
| **Replay** | Filtered packets are replayed using another socket. If sending fails, they are moved to a backup queue and retried twice. |
| **Error Handling** | Displays failed packet IDs and retries limited to 2 attempts. |
| **Display Functions** | Shows packet list, dissected layers, filtered packets, and replay status. |

### **Data Structures Used:**

#### **Custom Stack:**

#### Used for packet layer dissection.

**Operations:**

* push(layer)
* pop()
* peek()
* isEmpty()

#### **Custom Queue:**

#### Used for packet storage and replay management.

**Operations:**

* enqueue(packet)
* dequeue()
* isEmpty()
* size()

### **Algorithms Used:**

#### **Filtering Algorithm:**

for each packet in capturedPackets:

if packet.srcIP == filterSrc && packet.destIP == filterDest:

if packet.size < 1500:

filteredQueue.enqueue(packet)

#### **Replay Algorithm:**

for each packet in filteredQueue:

attempt = 0

while attempt < 2:

if send(packet):

break

else:

backupQueue.enqueue(packet)

attempt++

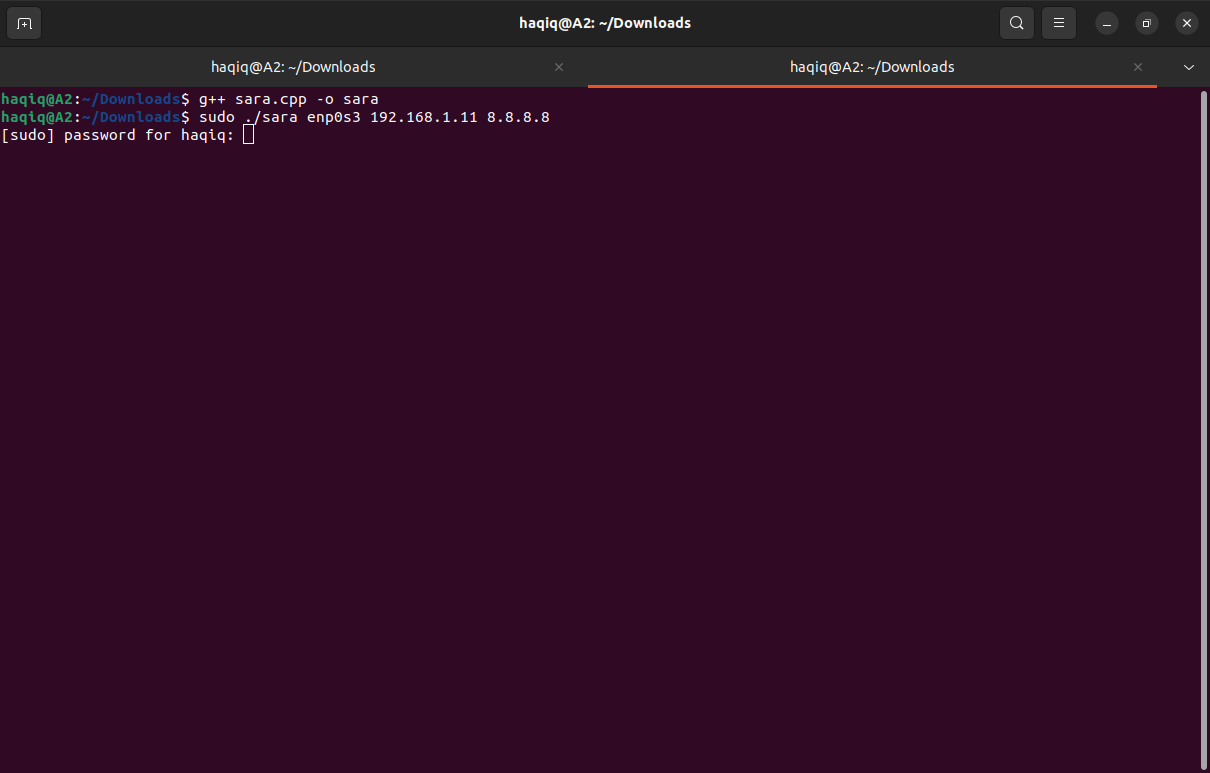
#### **Delay Calculation:**

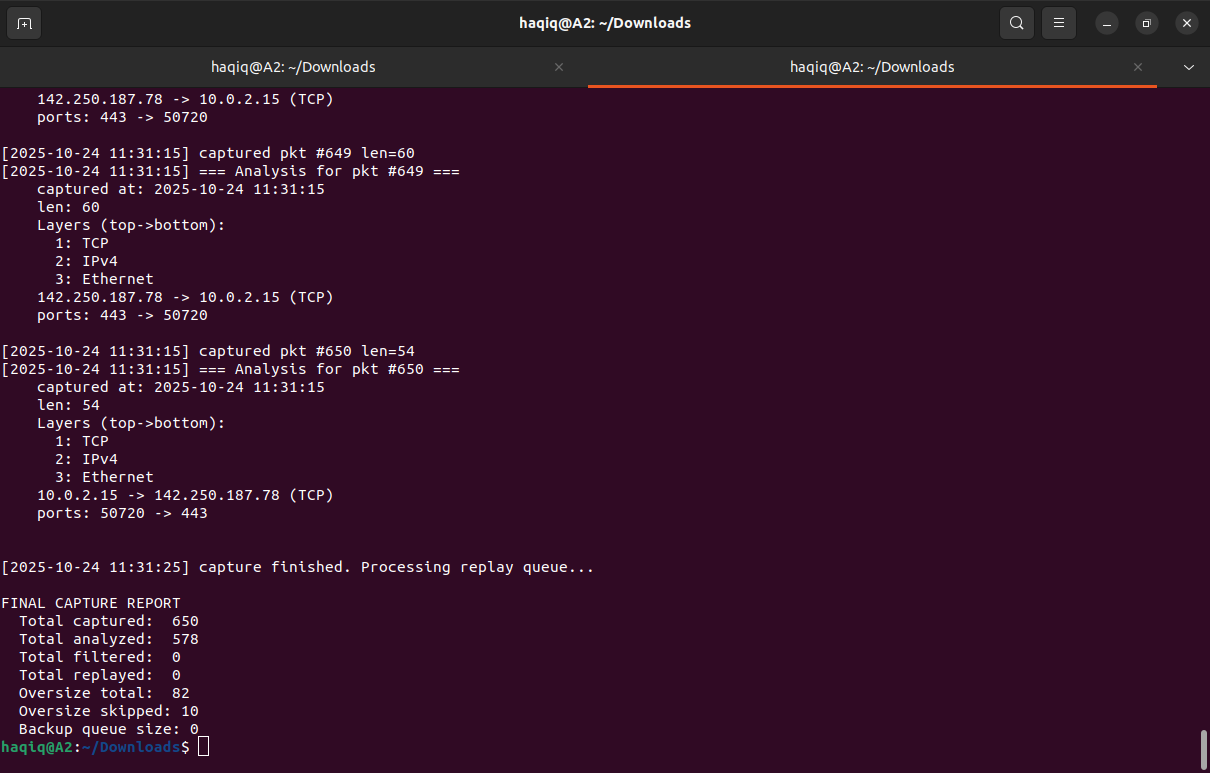
Total Delay = Packet Size / 1000 ms

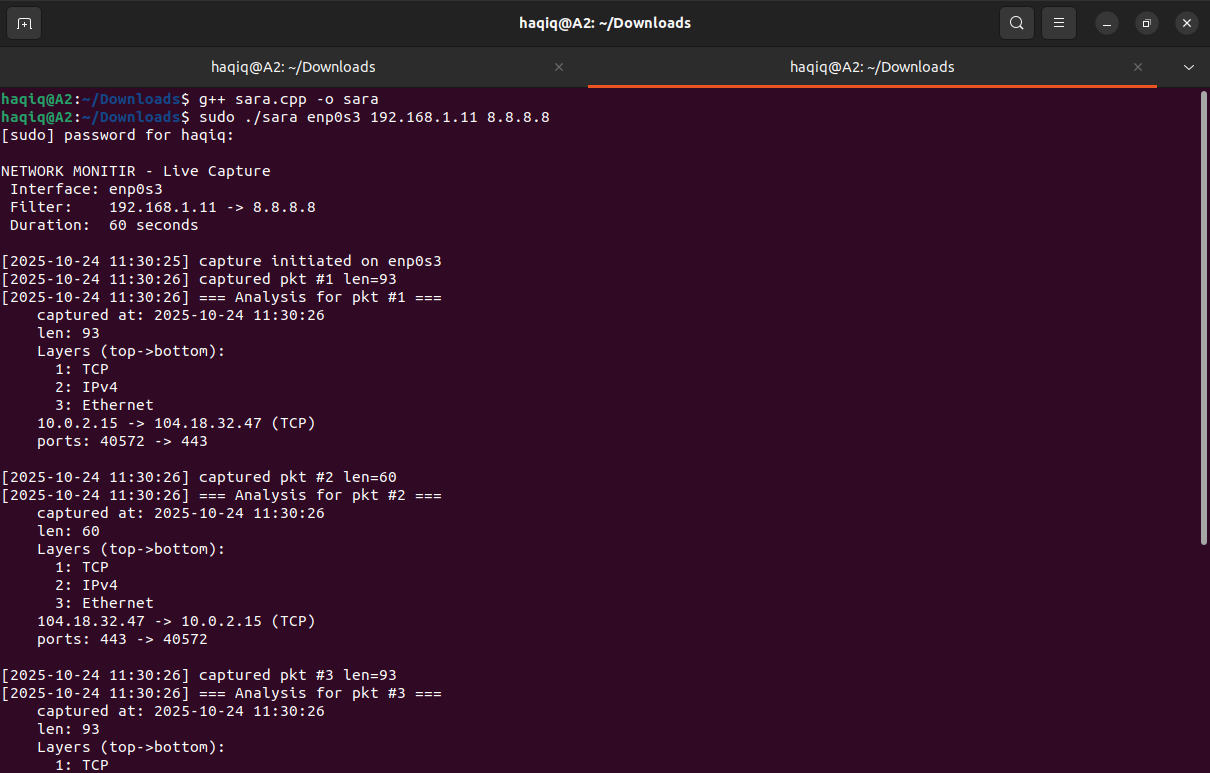
### **Execution Flow:**

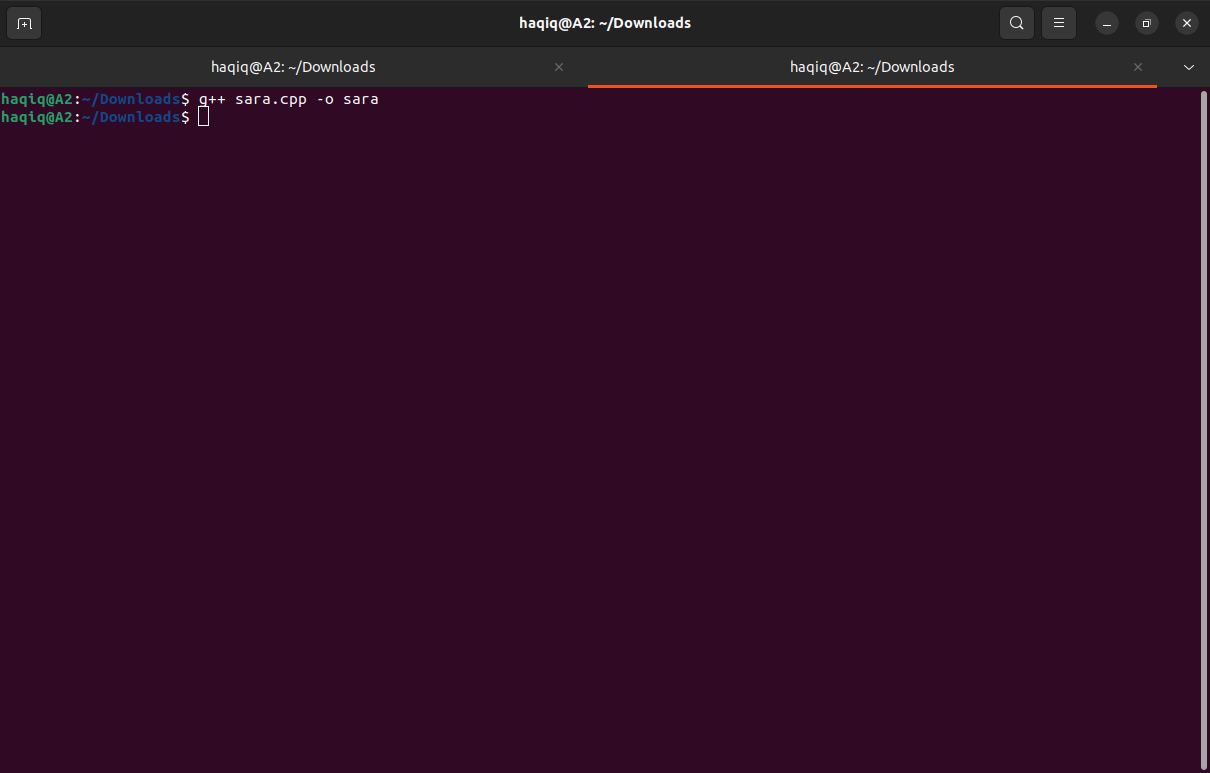
1. Open raw socket on interface
2. Capture packets continuously (1 min test case)
3. Parse layers using stack
4. Filter packets (by IPs)
5. Replay filtered packets
6. Handle failures → move to backup queue
7. Display dissected and filtered results

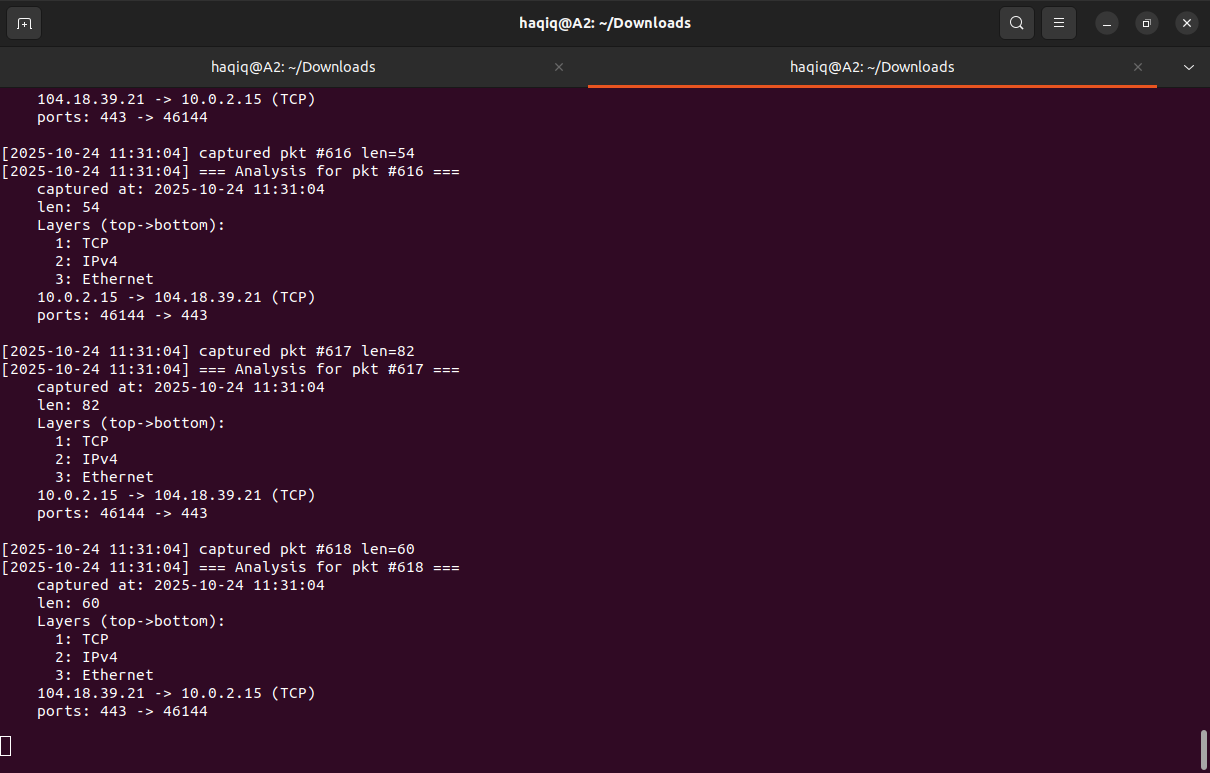
### **Output(ss):**

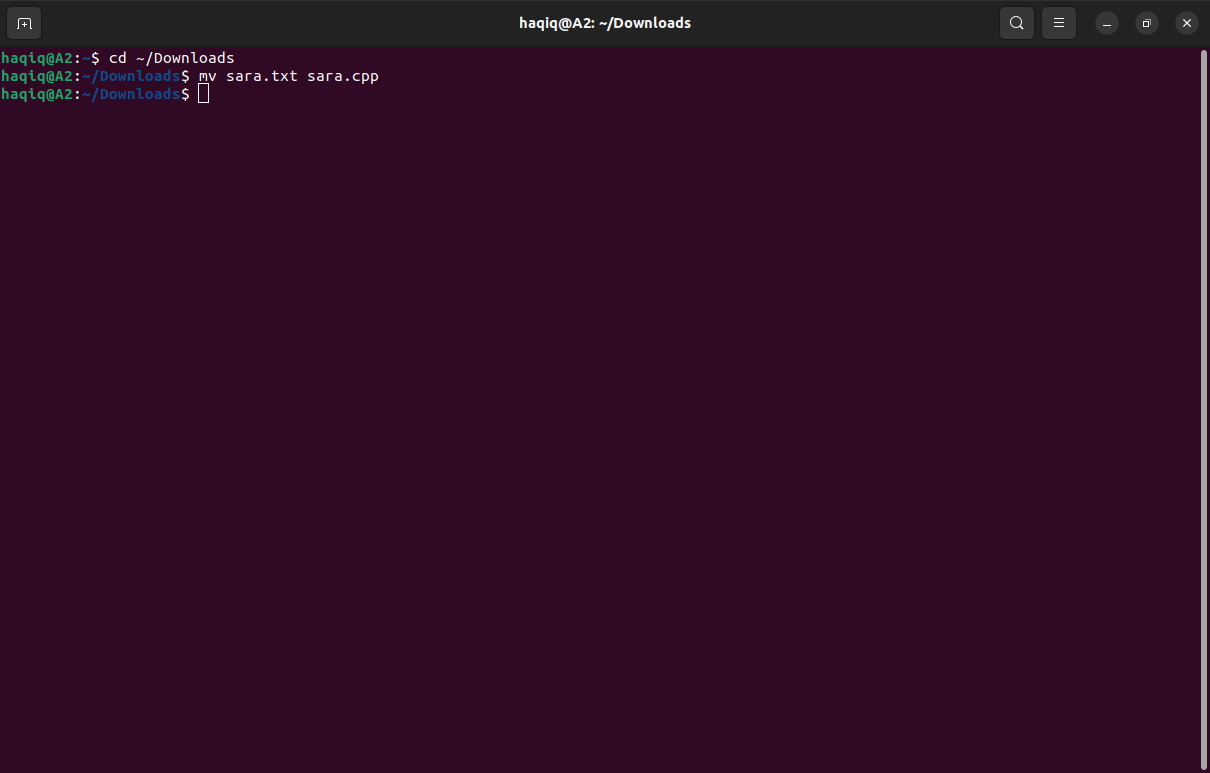


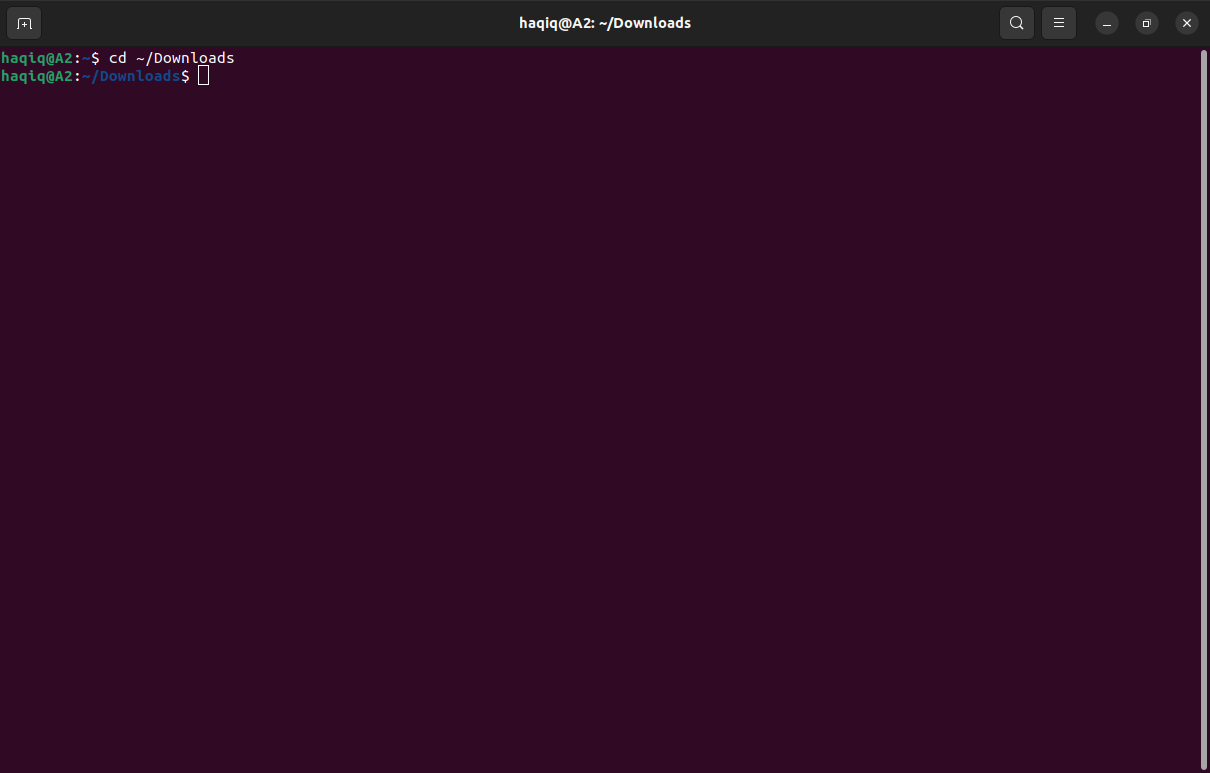












### **Challenges:**

* Managing raw socket permissions in Linux (requires sudo).
* Parsing packet headers manually without libraries.
* Ensuring stack and queue are memory safe.
* Handling continuous packet inflow efficiently.

### **Conclusion:**

The **Network Packet Monitor** successfully demonstrates the use of custom stacks and queues in handling real-time data streams. The project emphasizes algorithmic design, networking fundamentals, and practical system-level programming. This system can be further extended with GUI-based visualization and advanced protocol support (ICMP, ARP, etc.).

### **GitHub Submission Link:**

[sarafaw/DSA\_ASSIGNMENT2: C++ implementation of a Network Packet Monitor using custom Stack and Queue data structures. Captures, filters, dissects, and replays network packets in real time using raw sockets on Linux. Built for CS250 (Data Structures & Algorithms)](https://github.com/sarafaw/DSA_ASSIGNMENT2/tree/main)