### ****1. Introduction****

This assignment focuses on the practical application of **stacks** and **queues** in a real-time networking context. The goal was to implement a **Network Packet Analyzer and Replay System** using **C++** on a **Linux platform**. The system continuously captures packets from a specified network interface using **raw sockets**, stores them in a custom queue, parses them layer-by-layer using a **stack**, filters specific packets based on source and destination IP addresses, and replays them with error handling and retry mechanisms.

This project bridges theoretical understanding of data structures with real-world networking problems, strengthening concepts in **protocol parsing**, **traffic management**, and **algorithm design** under live network conditions.

### ****2. Assumptions****

1. The program runs on a **Linux** system with **root privileges** (required for raw sockets).
2. The network interface used for packet capture is eth0.
3. No external data structure libraries are used — only standard C++ STL for synchronization and basic containers.
4. The program runs continuously for 60 seconds by default, or until stopped manually using **Ctrl+C**.
5. Replay attempts are retried up to **two times** if a send operation fails.
6. The packet size threshold for filtering is **1500 bytes**.

### ****3. Objectives****

* Implement continuous packet capture using **raw sockets**.
* Use **custom stacks** for packet layer dissection (Ethernet, IPv4, IPv6, TCP, UDP).
* Use **custom queues** for packet management and replay lists.
* Filter and replay packets between specific IPs.
* Handle errors gracefully and provide meaningful display output.
* Demonstrate understanding of how stacks and queues apply in real-time systems.

### ****4. System Design and Data Structures****

#### ****4.1 Stacks****

A **custom stack** is implemented to manage **protocol layer dissection**.  
Each captured packet is pushed layer-by-layer (Ethernet → IP → Transport).  
During parsing, the stack allows sequential extraction and inspection of each header.

Example stack usage:

Push: EthernetHeader

Push: IPv4Header

Push: TCPHeader

Pop: TCPHeader

Pop: IPv4Header

Pop: EthernetHeader

#### ****4.2 Queues****

Two main **queues** manage packet flow:

1. **Capture Queue** – holds packets as they are captured from the network.
2. **Replay Queue** – holds filtered packets for replay or retry.

The queues use **thread-safe push/pop** operations and are implemented manually with synchronization primitives (mutex, condition\_variable).

### ****5. Implementation Details****

#### ****5.1 Packet Capture****

A **raw socket** (AF\_PACKET, SOCK\_RAW) is opened on interface eth0.  
Captured packets are assigned:

* **Packet ID**
* **Timestamp**
* **Raw data buffer**
* **Source and Destination IPs**

Each packet is wrapped in a Packet struct and added to the **capture queue**.

#### ****5.2 Packet Dissection****

The packet dissection thread uses the custom **stack** to extract protocol layers:

* Ethernet Header (MAC addresses)
* IPv4 or IPv6 Header (source/destination IPs)
* TCP/UDP Header (ports)

All parsing is done manually without external libraries. Each layer is printed to the console for inspection.

#### ****5.3 Filtering****

Packets are filtered by comparing their **source and destination IPs** to user-defined filters.  
Oversized packets (>1500 bytes) are skipped if they exceed a defined count threshold.  
Each filtered packet is placed into a **replay list**.

#### ****5.4 Replay and Error Handling****

Filtered packets are replayed using a separate **replay thread**.  
Each replayed packet estimates transmission delay as:

Total Delay (ms) = Packet Size / 1000

If a replay fails, it is moved to a **backup list** and retried up to **two times**.

#### ****5.5 Thread Management****

The system uses multiple threads:

* **Capture Thread:** Reads packets from raw socket.
* **Dissection Thread:** Parses layers via stack.
* **Filter Thread:** Selects packets by IP.
* **Replay Thread:** Replays filtered packets.

A global atomic flag running controls all loops.  
Threads exit gracefully when running is set to false.  
The program can be stopped anytime via **Ctrl+C**, using a SIGINT signal handler.

### ****6. Demonstration and Testing****

#### ****Test Setup****

* System: Ubuntu Linux
* Interface: eth0
* Duration: 60 seconds capture
* Permissions: Executed with sudo
* Commands:

g++ -pthread network\_monitor.cpp -o network\_monitor

sudo ./network\_monitor eth0

#### ****Test Steps****

1. Captured packets continuously for 1 minute.
2. Displayed live packet list with IDs, timestamps, and IPs.
3. Dissected multiple packets using custom stack.
4. Applied filtering between two sample IPs.
5. Replayed filtered packets with retry on failure.
6. Stopped capture using **Ctrl+C** safely.

#### ****Results****

* Successful continuous capture for 60 seconds.
* Correct protocol parsing and layer dissection.
* Filter and replay functionalities verified.
* Graceful exit and thread termination confirmed.

### ****8. Conclusion****

This assignment provided a strong practical connection between theoretical data structures and real-world networking systems.  
By implementing custom stacks and queues to manage live network data, this project deepened understanding of:

* Data structure operations under time constraints,
* Multi-thread synchronization, and
* Low-level packet handling using sockets.

Overall, the system fulfills all functional requirements and CLO outcomes.