**Network Packet Processing Module Analysis Report**

**Abstract**

This report provides a detailed analysis of two interdependent C++ header modules, packetcapture.h and parser.h, designed for low-level network data processing. It describes the working principles of each module, highlighting their respective features, and identifies the core Data Structures and Algorithms (DSA) used to achieve high-performance network monitoring and flow identification.

**1. The packetcapture.h Module: Data Link Layer Acquisition**

The PacketCapture module is engineered for **network packet sniffing** and **raw data acquisition**, relying on an external library like **Npcap** or **libpcap** for device interaction.

**1.1 Working Principle**

1. **Initialization:** The initialize method establishes the capture session by accepting the network interface name (device\_name) and an optional **Berkeley Packet Filter (BPF)** string. It is responsible for opening the device and applying the filter rules.
2. **Data Structure (RawPacket):** When a frame is received, it is immediately encapsulated into a RawPacket struct, which contains essential metadata:
   * ts\_us: Microsecond-precision timestamp.
   * len: Length of the captured frame.
   * data: The full, raw byte array of the packet.
3. **Capture Loop:** The capture\_loop executes the continuous, **blocking** process of receiving frames. For every frame captured, it constructs a RawPacket and delivers it to the registered handler.
4. **Control:** The stop method ensures a **thread-safe** way to signal termination of the capture\_loop by manipulating the volatile bool running flag.

**1.2 Features and DSA Usage**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Description** | **DSA/Concept Used** |
| **Raw Packet Storage** | Stores the entire frame's binary data efficiently. | **Dynamic Array/Buffer:** std::vector<uint8\_t> is used in RawPacket::data to handle the variable-length raw byte stream. |
| **Event Handling** | Allows decoupled processing of packets by external functions. | **Function Object/Callback:** PacketHandler (an alias for std::function) implements the **Observer Pattern**, notifying clients when new data is available. |
| **Concurrency Control** | Manages the safe stopping of the capture thread. | **Simple Flag:** The volatile bool running provides a memory-visible state for thread synchronization. |

**2. The parser.h Module: Protocol Decapsulation and Flow Identification**

The parser.h module takes the raw bytes provided by PacketCapture and performs the critical task of **decoding protocol headers** to extract application-level data and identify network flows.

**2.1 Working Principle**

1. **Flow Identification (FiveTuple):** The primary data structure for flow identification is the FiveTuple, which uniquely defines a network conversation using source/destination IP addresses, source/destination ports, and the transport layer protocol. This structure serves as a fundamental **key** for subsequent analysis (e.g., connection tracking).
2. **Decoded Structure (ParsedPacket):** This structure aggregates the results of the parsing process, containing the FiveTuple, the original timestamp, and most importantly, a **pointer and length** (payload, payload\_len) to the start of the application-layer data within the raw packet.
3. **Parsing Algorithm:** The core function, parse\_ethernet\_ipv4, executes a **sequential traversal** algorithm. It inspects the bytes of the raw packet to:
   * Verify the Ethernet Frame Type (must be IPv4).
   * Determine the length of the IPv4 header.
   * Extract the IPs, ports, and transport protocol from the headers.
   * Calculate the exact starting position of the application payload.

**2.2 Features and DSA Usage**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Description** | **DSA/Concept Used** |
| **Flow Key** | Structure used to uniquely identify any network connection or stream. | **Record Data Structure:** struct FiveTuple is an aggregate data type that acts as a composite **Key** for network flow indexing. |
| **Payload Access** | Provides highly efficient access to the application data without memory duplication. | **Pointer/Reference:** Using const uint8\_t\* payload ensures a **Zero-Copy** approach, where the parser points directly to the data within the original raw buffer. |
| **Header Decoding** | Logic to interpret protocol specifications from the raw byte stream. | **Sequential Access and Bitwise Operations:** The parser performs **algorithmic traversal** over the byte array, using bitmasks and shifts to correctly extract fields (e.g., flags, header lengths, protocol identifiers). |
| **Network-to-Host Conversion** | Standard functions to ensure multi-byte integers are interpreted correctly across different system architectures (endianness). | **Standard Algorithms:** Functions like ntohl (Network To Host Long) are necessary for correctly processing IP addresses and port numbers. |