Network Monitoring System - Comprehensive Architecture Document

Table of Contents

Executive Summary

The Network Monitoring System is a comprehensive real-time network traffic analysis and monitoring application built using C++20, Qt6 for the GUI, and various networking libraries. The system captures network packets, analyzes traffic patterns, stores historical data, and provides real-time visualization of network activity.

Key Features

Real-time packet capture using libpcap
Protocol analysis (TCP, UDP, HTTP, HTTPS, DNS, DHCP, ARP)
Traffic statistics and bandwidth monitoring
SQLite-based data persistence
Qt6-based graphical user interface
Configurable filtering using BPF expressions
Plugin architecture for extensibility
gRPC API for external integrations

System Overview

Technology Stack

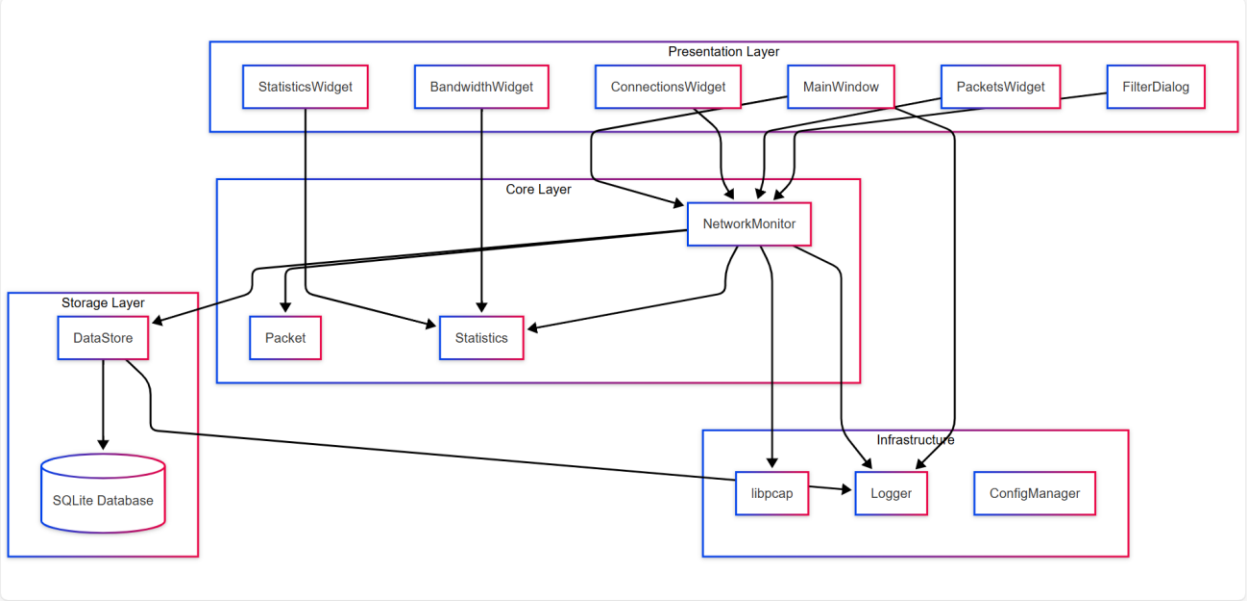
Language: C++20
GUI Framework: Qt6 (Core, Gui, Widgets, Charts)
Packet Capture: libpcap
Database: SQLite3
Networking: Boost.Asio
Security: OpenSSL
API: gRPC with Protocol Buffers
Build System: CMake 3.15+
Logging: Custom Logger implementation

Architecture Style

The system follows a multi-threaded, event-driven architecture with clear separation of concerns:

Presentation Layer: Qt6 GUI components
Business Logic Layer: Core monitoring and analysis components
Data Access Layer: SQLite storage and data management
Infrastructure Layer: Packet capture and system utilities

Architecture Overview




```
graph TB
    subgraph "Presentation Layer"
        MW[MainWindow]
        SW[StatisticsWidget]
        CW[ConnectionsWidget]
        PW[PacketsWidget]
        BW[BandwidthWidget]
        FD[FilterDialog]
    end

    subgraph "Core Layer"
        NM[NetworkMonitor]
        PKT[Packet]
        STAT[Statistics]
    end

    subgraph "Storage Layer"
        DS[DataStore]
        DB[(SQLite Database)]
    end

    subgraph "Infrastructure"
```

```
    PCAP[libpcap]
    LOG[Logger]
    CFG[ConfigManager]
end

MW --> NM
SW --> STAT
CW --> NM
PW --> NM
BW --> STAT
FD --> NM

NM --> PKT
NM --> STAT
NM --> DS
NM --> PCAP

DS --> DB

NM --> LOG
DS --> LOG
MW --> LOG
```

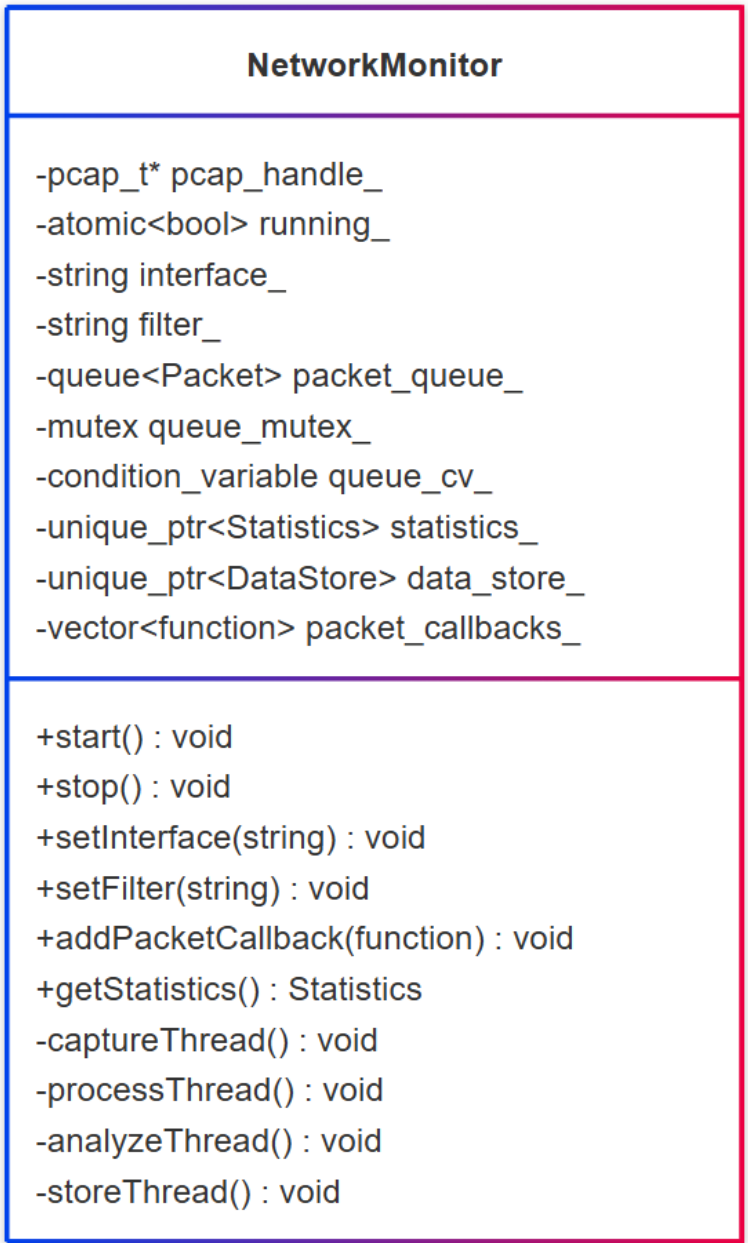
Component Architecture

1. Core Components

NetworkMonitor

The central component responsible for:

- Managing packet capture through libpcap
- Coordinating multiple processing threads
- Distributing packets to analysis components
- Managing filters and capture settings

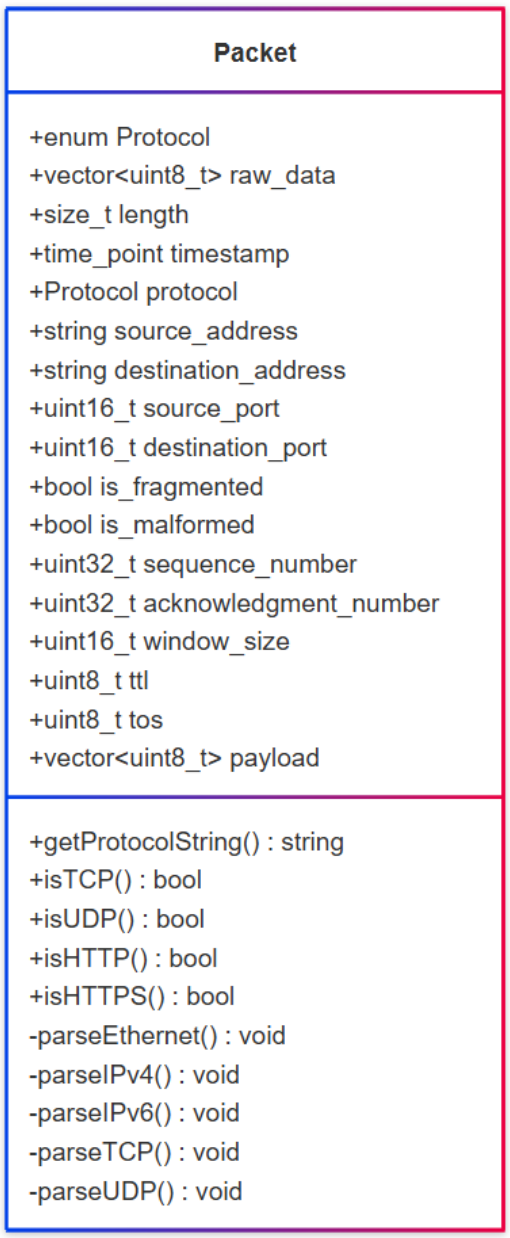



```
classDiagram
    class NetworkMonitor {
        -pcap_t* pcap_handle_
        -atomic~bool~ running_
        -string interface_
        -string filter_
        -queue~Packet~ packet_queue_
        -mutex queue_mutex_
        -condition_variable queue_cv_
        -unique_ptr~Statistics~ statistics_
        -unique_ptr~DataStore~ data_store_
```

```
        -vector~function~ packet_callbacks_
        +start() void
        +stop() void
        +setInterface(string) void
        +setFilter(string) void
        +addPacketCallback(function) void
        +getStatistics() Statistics
        -captureThread() void
        -processThread() void
        -analyzeThread() void
        -storeThread() void
    }
```

Packet

Data structure representing captured network packets:

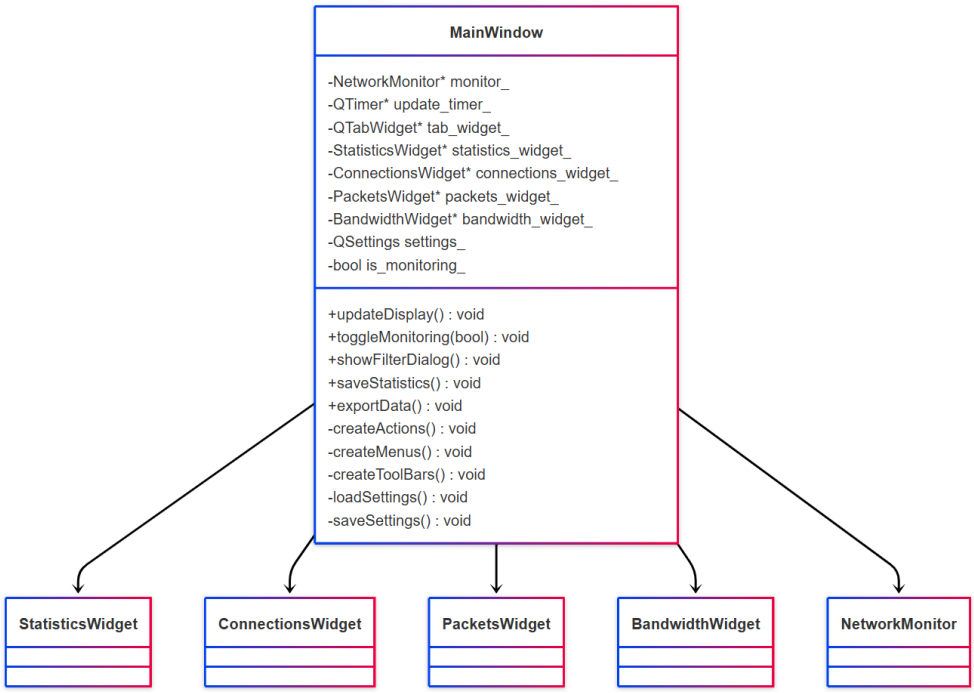



```
classDiagram
    class Packet {
        +enum Protocol
        +vector~uint8_t~ raw_data
        +size_t length
        +time_point timestamp
        +Protocol protocol
        +string source_address
        +string destination_address
        +uint16_t source_port
```

```
        +uint16_t destination_port
        +bool is_fragmented
        +bool is_malformed
        +uint32_t sequence_number
        +uint32_t acknowledgment_number
        +uint16_t window_size
        +uint8_t ttl
        +uint8_t tos
        +vector~uint8_t~ payload
        +getProtocolString() string
        +isTCP() bool
        +isUDP() bool
        +isHTTP() bool
        +isHTTPS() bool
        -parseEthernet() void
        -parseIPv4() void
        -parseIPv6() void
        -parseTCP() void
        -parseUDP() void
    }
```

2. GUI Components

The GUI layer is built using Qt6 and follows the Model-View pattern:




```
classDiagram
    class MainWindow {
        -NetworkMonitor* monitor_
        -QTimer* update_timer_
        -QTabWidget* tab_widget_
        -StatisticsWidget* statistics_widget_
        -ConnectionsWidget* connections_widget_
        -PacketsWidget* packets_widget_
        -BandwidthWidget* bandwidth_widget_
        -QSettings settings_
        -bool is_monitoring_
        +updateDisplay() void
        +toggleMonitoring(bool) void
        +showFilterDialog() void
        +saveStatistics() void
        +exportData() void
        -createActions() void
        -createMenus() void
        -createToolBars() void
        -loadSettings() void
        -saveSettings() void
    }
```

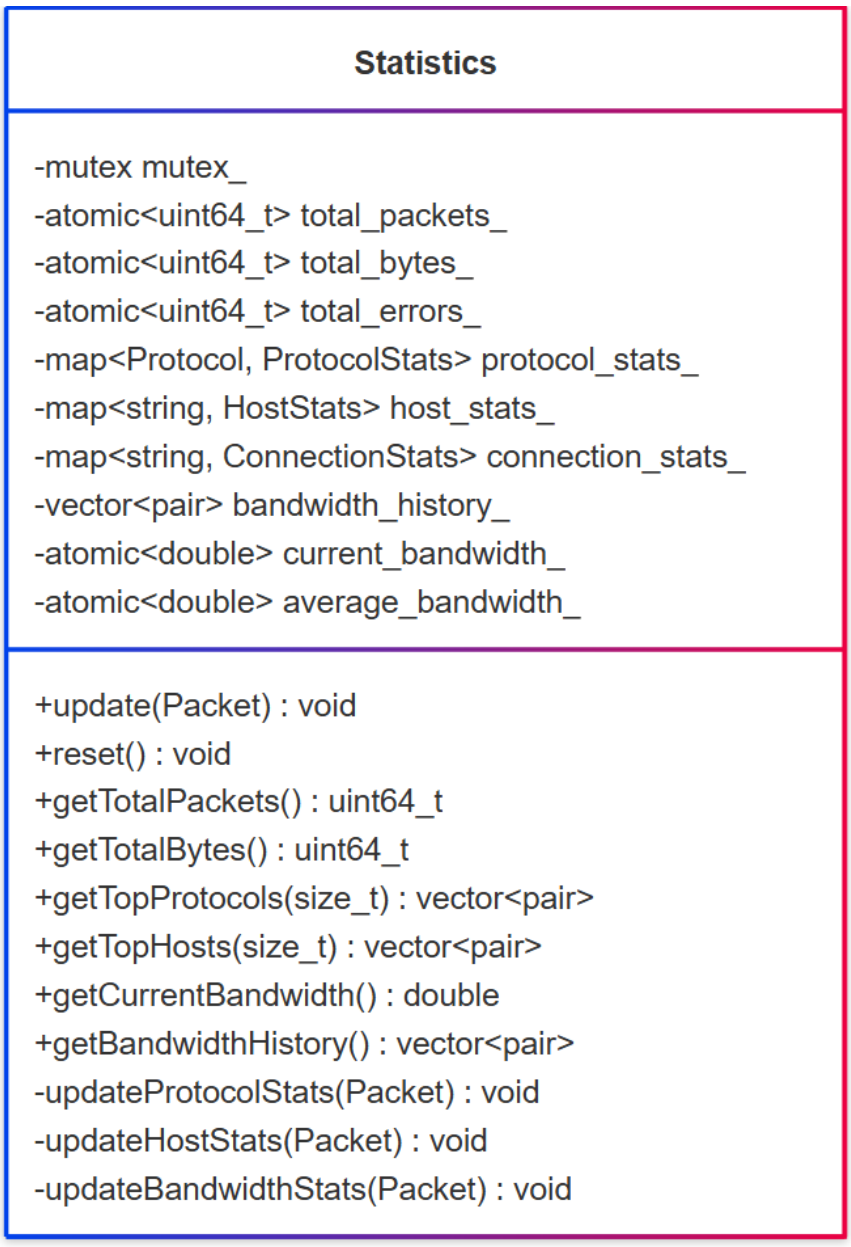
```
    MainWindow --> StatisticsWidget
    MainWindow --> ConnectionsWidget
    MainWindow --> PacketsWidget
    MainWindow --> BandwidthWidget
    MainWindow --> NetworkMonitor
```

3. Storage Components


```
classDiagram
    class DataStore {
        -sqlite3* db_
        -string db_path_
        -atomic~bool~ running_
        -thread store_thread_
        -queue~Packet~ packet_queue_
```

```
        -mutex queue_mutex_
        -condition_variable queue_cv_
        +store(Packet) void
        +flush() void
        +close() void
        +getPacketsByProtocol(Protocol, size_t) vector~Packet~
        +getPacketsByHost(string, size_t) vector~Packet~
        +getPacketsByTimeRange(time_point, time_point, size_t)
vector~Packet~
        +getPacketCount() uint64_t
        +getByteCount() uint64_t
        -initializeDatabase() void
        -createTables() void
        -storeThread() void
        -batchInsert() void
    }
```

4. Analysis Components

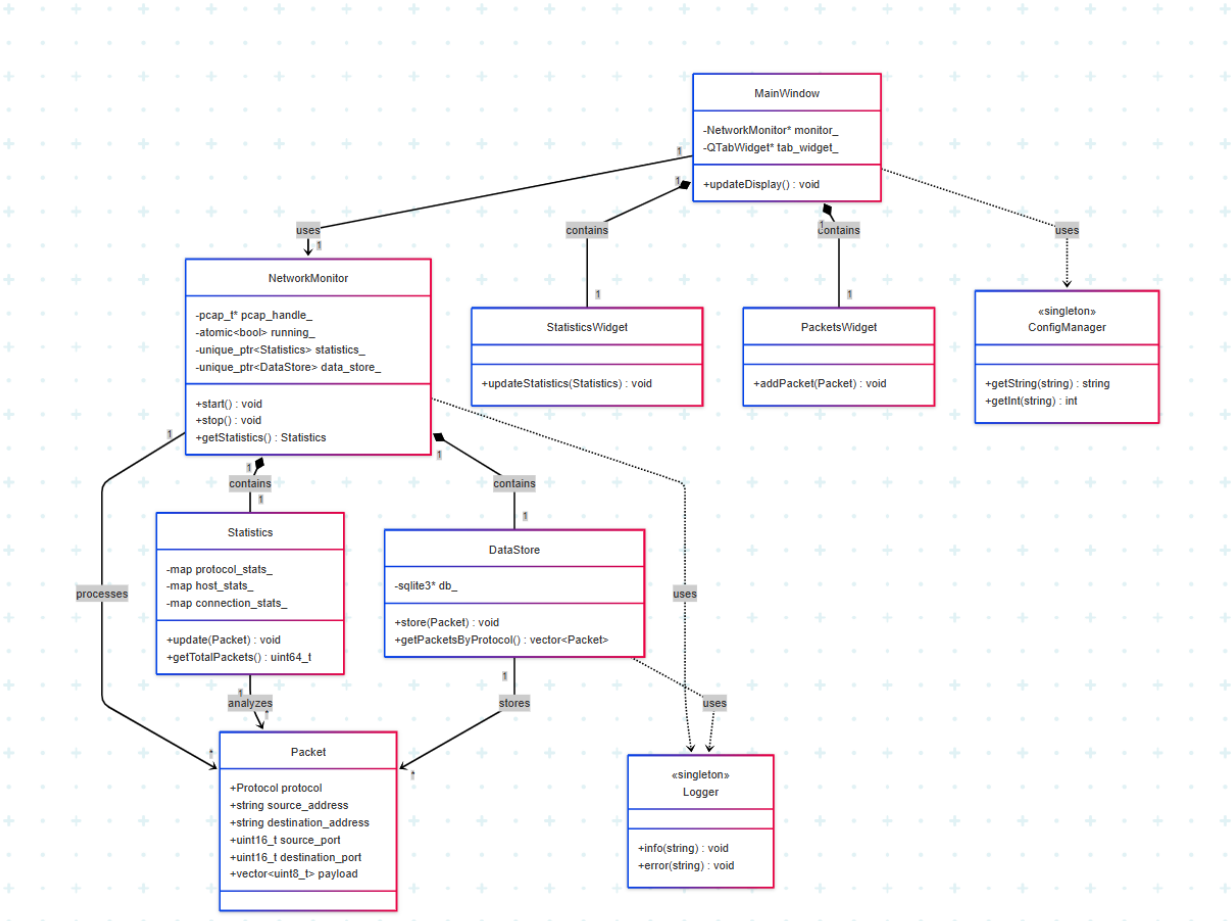



```
classDiagram
    class Statistics {
        -mutex mutex_
        -atomic~uint64_t~ total_packets_
        -atomic~uint64_t~ total_bytes_
        -atomic~uint64_t~ total_errors_
        -map~Protocol, ProtocolStats~ protocol_stats_
        -map~string, HostStats~ host_stats_
```

```
    -map~string, ConnectionStats~ connection_stats_
    -vector~pair~ bandwidth_history_
    -atomic~double~ current_bandwidth_
    -atomic~double~ average_bandwidth_
    +update(Packet) void
    +reset() void
    +getTotalPackets() uint64_t
    +getTotalBytes() uint64_t
    +getTopProtocols(size_t) vector~pair~
    +getTopHosts(size_t) vector~pair~
    +getCurrentBandwidth() double
    +getBandwidthHistory() vector~pair~
    -updateProtocolStats(Packet) void
    -updateHostStats(Packet) void
    -updateBandwidthStats(Packet) void
}
```

Class Diagrams and Object Relationships

Complete System Class Diagram

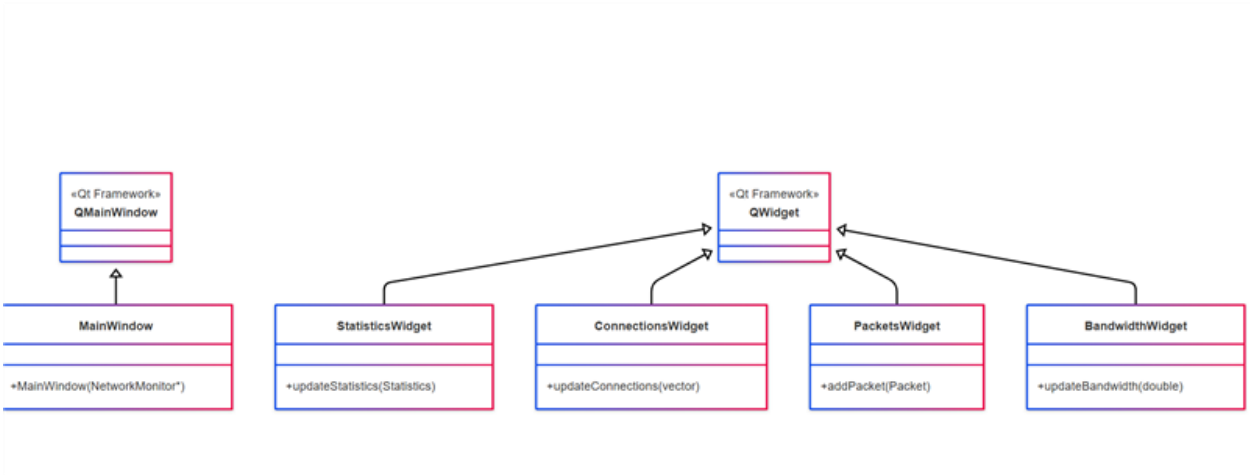



```
classDiagram
    %% Core Classes
    class NetworkMonitor {
        -pcap_t* pcap_handle_
        -atomic~bool~ running_
        -unique_ptr~Statistics~ statistics_
        -unique_ptr~DataStore~ data_store_
        +start() void
        +stop() void
        +getStatistics() Statistics
    }
```

```
class Packet {
    +Protocol protocol
    +string source_address
    +string destination_address
    +uint16_t source_port
    +uint16_t destination_port
    +vector~uint8_t~ payload
}

class Statistics {
    -map protocol_stats_
    -map host_stats_
    -map connection_stats_
    +update(Packet) void
    +getTotalPackets() uint64_t
}

class DataStore {
    -sqlite3* db_
    +store(Packet) void
    +getPacketsByProtocol() vector~Packet~
}

%% GUI Classes
class MainWindow {
    -NetworkMonitor* monitor_
    -QTabWidget* tab_widget_
    +updateDisplay() void
}

class StatisticsWidget {
    +updateStatistics(Statistics) void
}

class PacketsWidget {
    +addPacket(Packet) void
}

%% Utility Classes
```

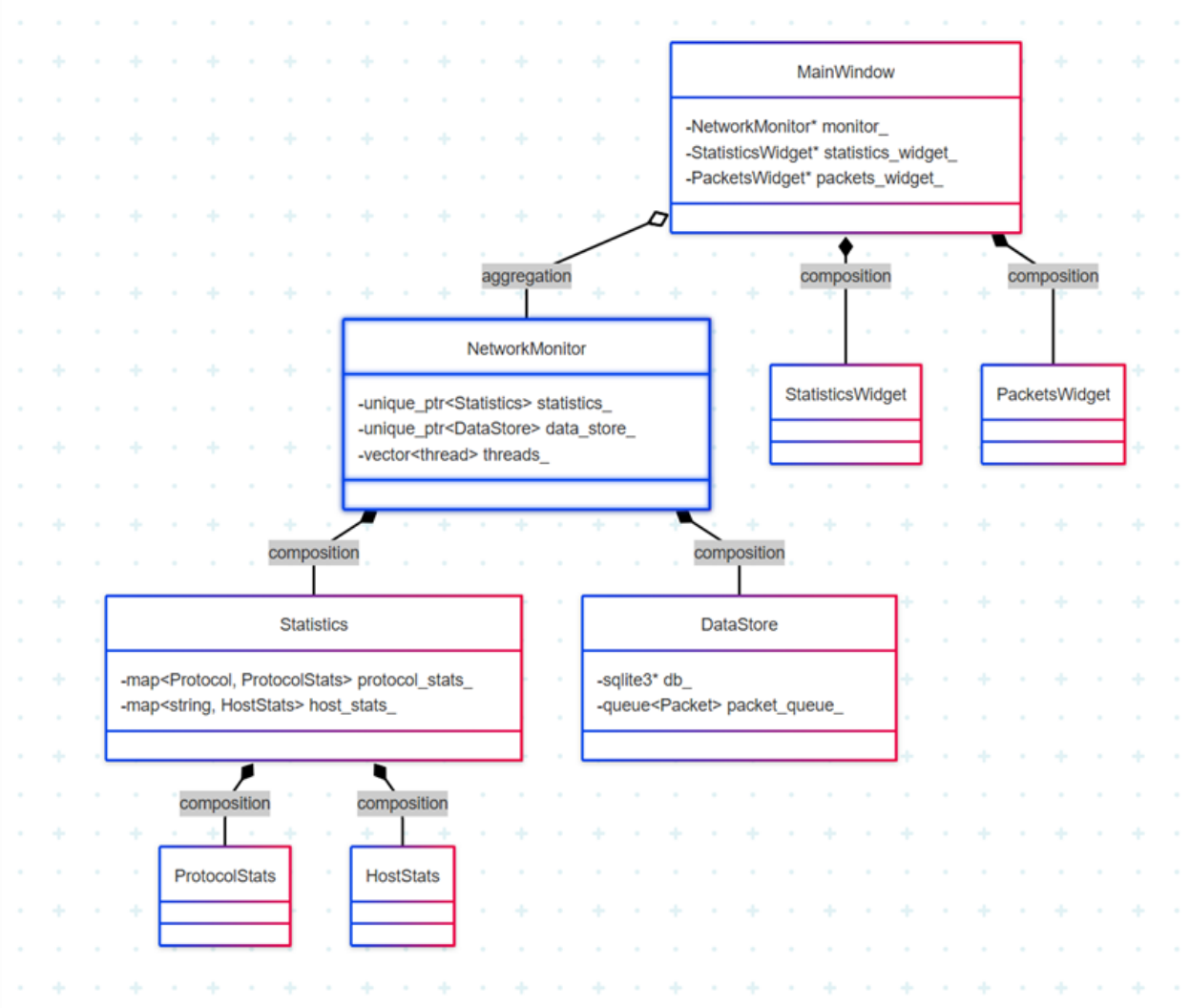
```
class Logger {
    <<singleton>>
    +info(string) void
    +error(string) void
}

class ConfigManager {
    <<singleton>>
    +getString(string) string
    +getInt(string) int
}

%% Relationships
NetworkMonitor "1" *-- "1" Statistics : contains
NetworkMonitor "1" *-- "1" DataStore : contains
NetworkMonitor "1" --> "*" Packet : processes
MainWindow "1" --> "1" NetworkMonitor : uses
MainWindow "1" *-- "1" StatisticsWidget : contains
MainWindow "1" *-- "1" PacketsWidget : contains
Statistics "1" --> "*" Packet : analyzes
DataStore "1" --> "*" Packet : stores
NetworkMonitor ..> Logger : uses
DataStore ..> Logger : uses
MainWindow ..> ConfigManager : uses
```

Inheritance Hierarchy



```
classDiagram
    class QMainWindow {
        <<Qt Framework>>
    }

    class QWidget {
        <<Qt Framework>>
    }

    class MainWindow {
        +MainWindow(NetworkMonitor*)
    }

    class StatisticsWidget {
        +updateStatistics(Statistics)
    }

    class ConnectionsWidget {
        +updateConnections(vector)
    }

    class PacketsWidget {
        +addPacket(Packet)
    }

    class BandwidthWidget {
        +updateBandwidth(double)
    }

    QMainWindow <|-- MainWindow
    QWidget <|-- StatisticsWidget
    QWidget <|-- ConnectionsWidget
    QWidget <|-- PacketsWidget
    QWidget <|-- BandwidthWidget
```

Composition and Aggregation Relationships



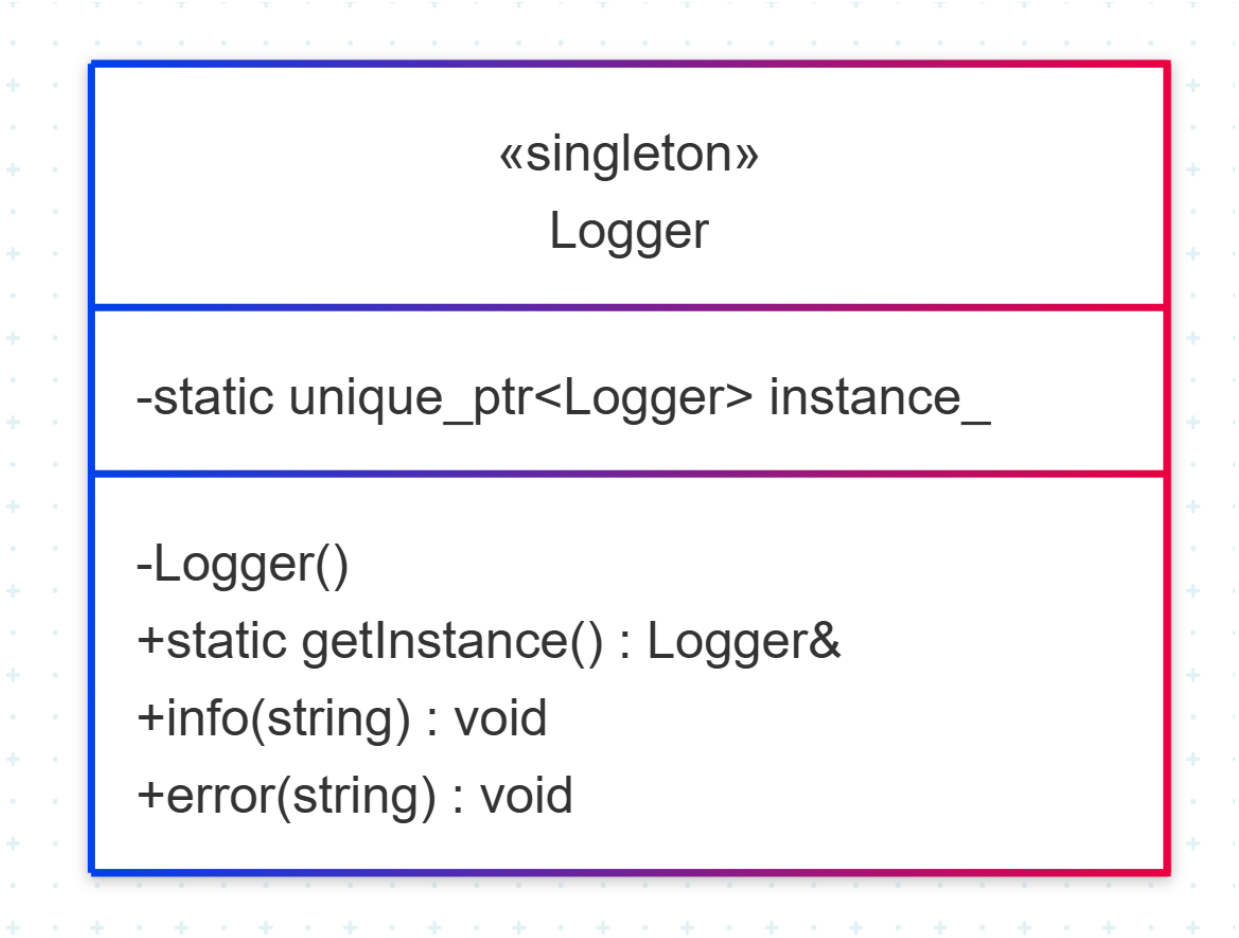

```
classDiagram
    class NetworkMonitor {
        -unique_ptr~Statistics~ statistics_
        -unique_ptr~DataStore~ data_store_
        -vector~thread~ threads_
    }

    class Statistics {
        -map~Protocol, ProtocolStats~ protocol_stats_
        -map~string, HostStats~ host_stats_
    }
```

```
class DataStore {
    -sqlite3* db_
    -queue~Packet~ packet_queue_
}

class MainWindow {
    -NetworkMonitor* monitor_
    -StatisticsWidget* statistics_widget_
    -PacketsWidget* packets_widget_
}

NetworkMonitor *-- Statistics : composition
NetworkMonitor *-- DataStore : composition
Statistics *-- ProtocolStats : composition
Statistics *-- HostStats : composition
MainWindow o-- NetworkMonitor : aggregation
MainWindow *-- StatisticsWidget : composition
MainWindow *-- PacketsWidget : composition
```

Design Patterns

1. Singleton Pattern

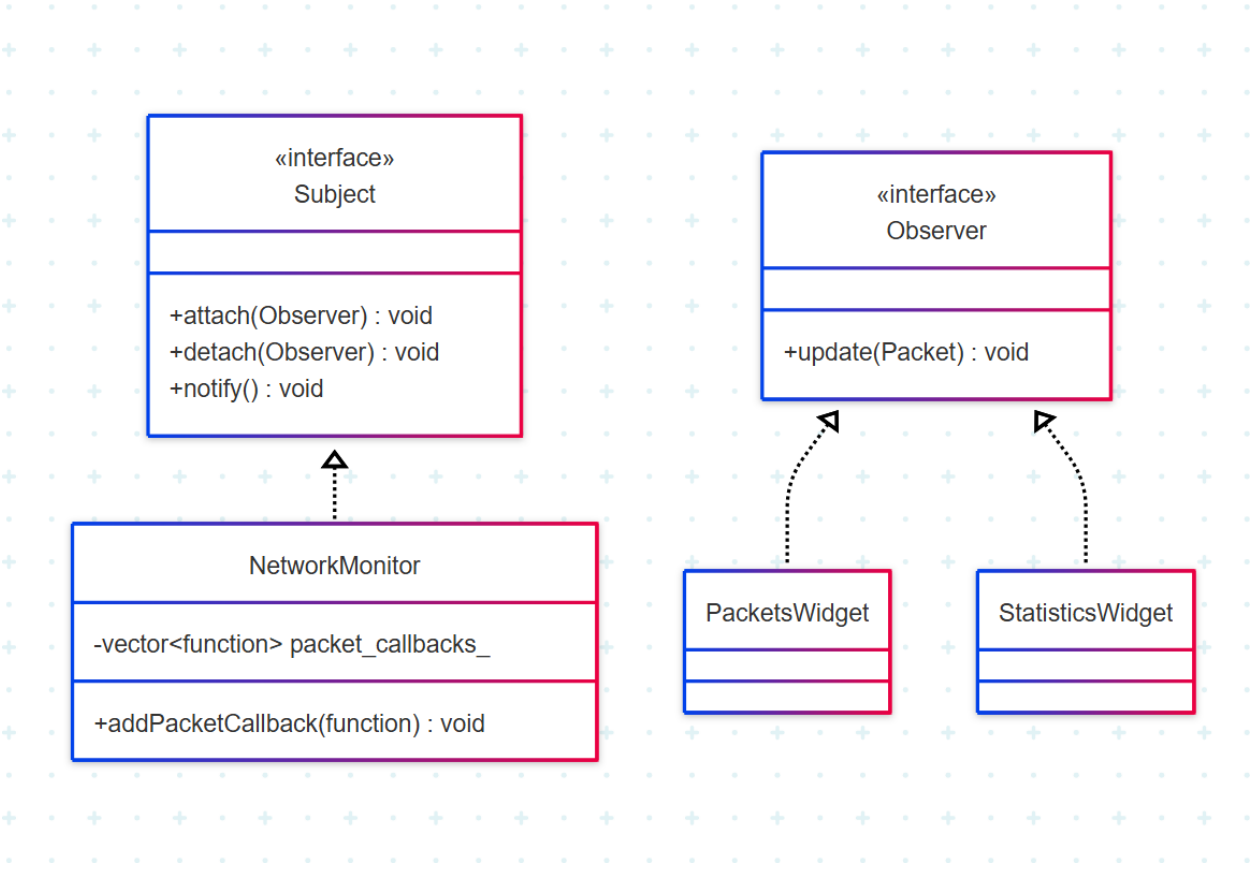
Used for Logger and ConfigManager to ensure single instances:




```
classDiagram
    class Logger {
        <<singleton>>
        -static unique_ptr~Logger~ instance_
        -Logger()
        +static getInstance() Logger&
        +info(string) void
        +error(string) void
    }
```

2. Observer Pattern

Implemented through packet callbacks:

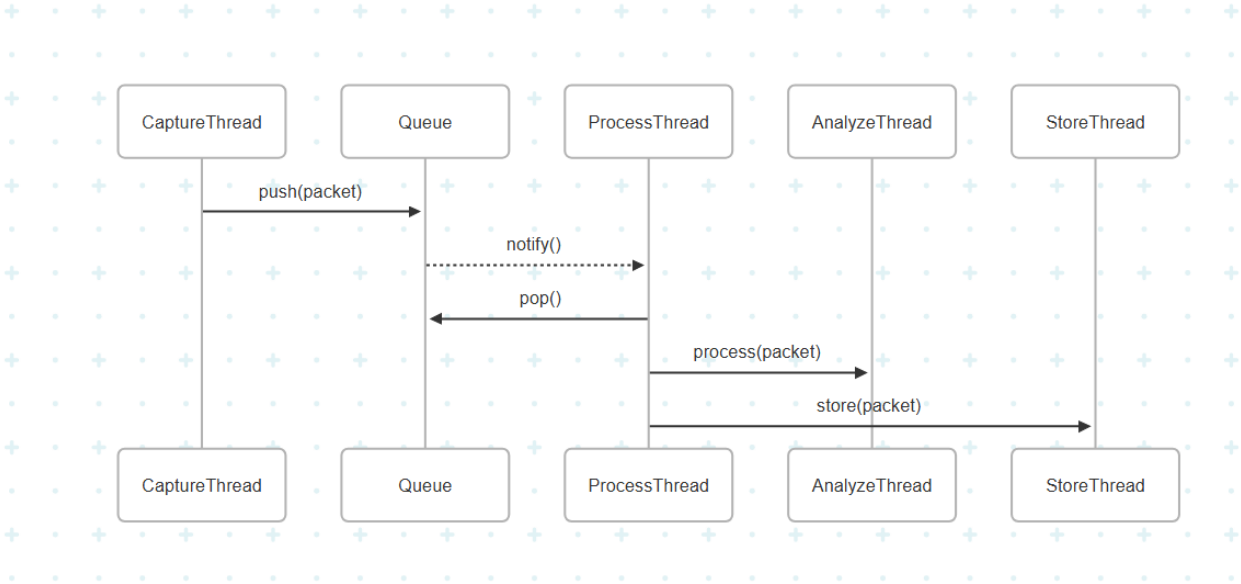



```
classDiagram
    class Subject {
        <<interface>>
        +attach(Observer) void
        +detach(Observer) void
        +notify() void
    }

    class NetworkMonitor {
        -vector~function~ packet_callbacks_
        +addPacketCallback(function) void
    }

    class Observer {
        <<interface>>
        +update(Packet) void
    }
```

```
    Subject <|.. NetworkMonitor
    Observer <|.. PacketsWidget
    Observer <|.. StatisticsWidget
```

3. Producer-Consumer Pattern

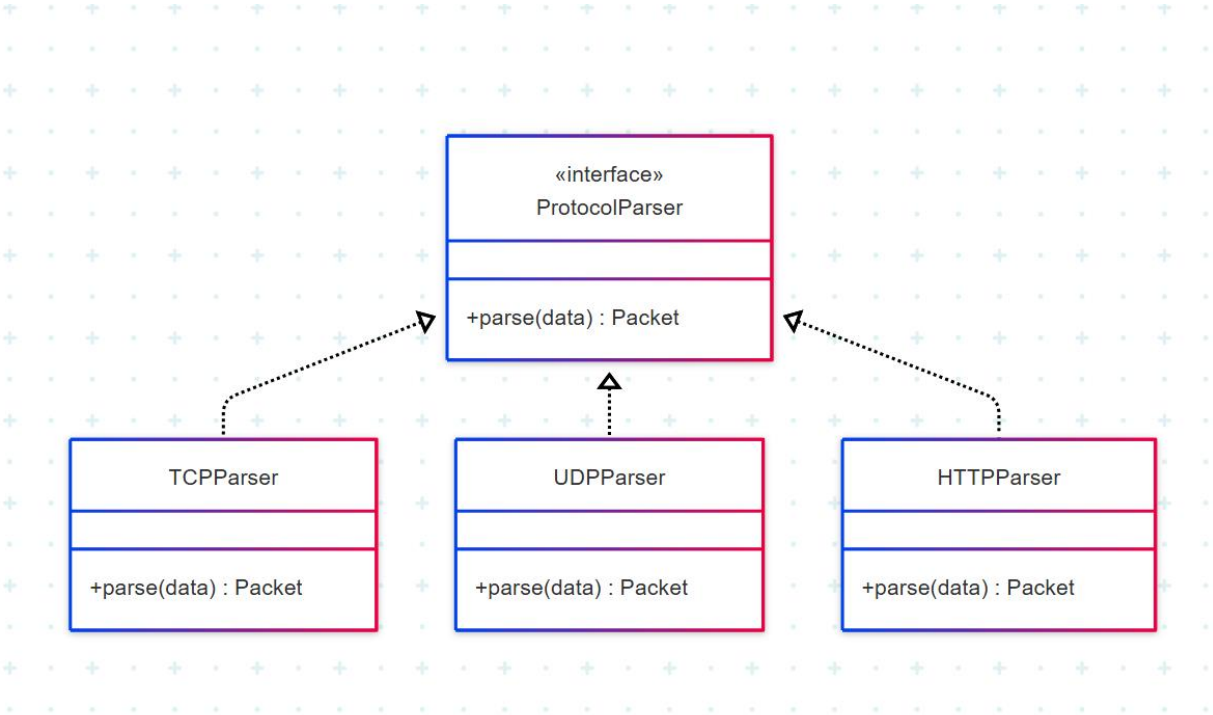
Used for packet processing with thread-safe queues:




```
sequenceDiagram
    participant CaptureThread
    participant Queue
    participant ProcessThread
    participant AnalyzeThread
    participant StoreThread

    CaptureThread->>Queue: push(packet)
    Queue-->>ProcessThread: notify()
    ProcessThread->>Queue: pop()
    ProcessThread->>AnalyzeThread: process(packet)
    ProcessThread->>StoreThread: store(packet)
```

4. Strategy Pattern

For protocol parsing:




```
classDiagram
    class ProtocolParser {
        <<interface>>
        +parse(data) Packet
    }

    class TCPParser {
        +parse(data) Packet
    }

    class UDPParser {
        +parse(data) Packet
    }

    class HTTPParser {
        +parse(data) Packet
    }
```

```
ProtocolParser <|.. TCPParser
ProtocolParser <|.. UDPParser
ProtocolParser <|.. HTTPParser
```

External System Dependencies

1. Direct Dependencies

libpcap (Packet Capture)

Purpose: Low-level network packet capture
Integration: Direct API calls through pcap.h
Authentication: None (requires root/admin privileges)
Data Flow: Inbound raw packet data

SQLite3 (Database)

Purpose: Local data persistence
Integration: Direct API calls through sqlite3.h
Authentication: None (local file-based)
Data Flow: Bidirectional (store and retrieve packets)

Qt6 Framework

Purpose: GUI framework and event handling
Integration: Inheritance and composition
Components Used: Core, Gui, Widgets, Charts
Data Flow: Event-driven UI updates

2. Build Dependencies

Boost Libraries

Components: system, filesystem
Purpose: Cross-platform system operations
Integration: Header-only and linked libraries

OpenSSL

Purpose: SSL/TLS packet analysis
Integration: Linked libraries for crypto operations

gRPC & Protocol Buffers

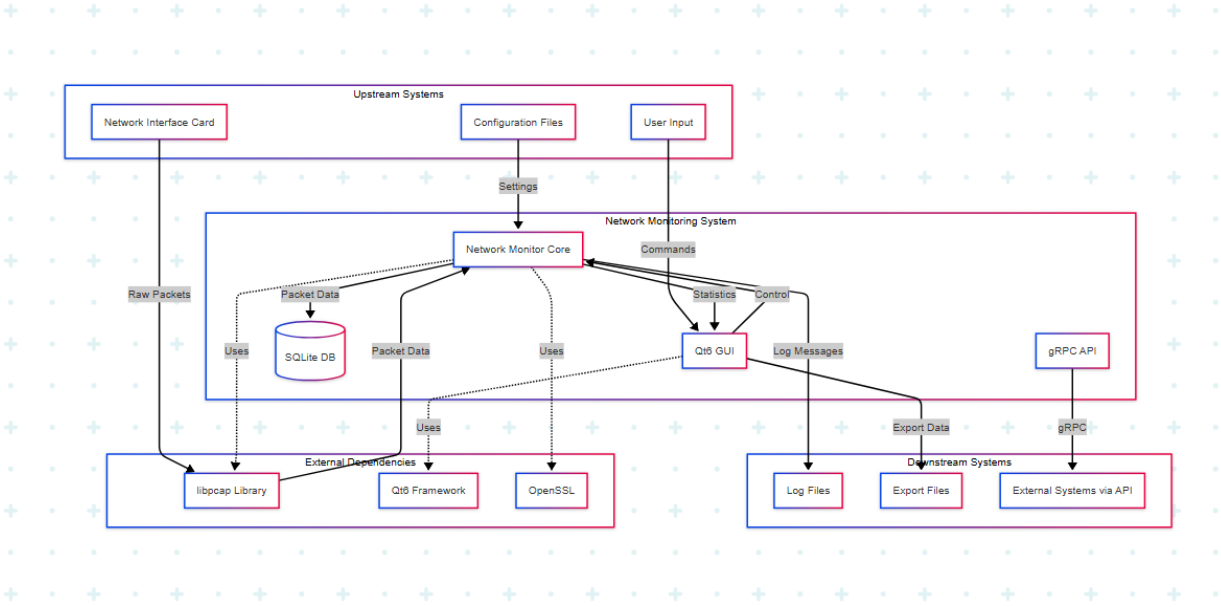
Purpose: API for external system integration
Integration: Code generation and runtime libraries
Note: API implementation not found in current codebase (planned feature)

3. System Requirements

Operating System Integration

Network Interfaces: Direct access via libpcap
Privileges: Requires elevated permissions for packet capture
File System: Read/write access for database and logs

System Context Diagram




```
graph TB
    subgraph "Network Monitoring System"
```

```
    NMS[Network Monitor Core]
    GUI[Qt6 GUI]
    DB[(SQLite DB)]
    API[gRPC API]
end

subgraph "Upstream Systems"
    NIC[Network Interface Card]
    CFG[Configuration Files]
    USR[User Input]
end

subgraph "Downstream Systems"
    LOG[Log Files]
    EXP[Export Files]
    EXT[External Systems via API]
end

subgraph "External Dependencies"
    PCAP[libpcap Library]
    QT[Qt6 Framework]
    SSL[OpenSSL]
end

%% Upstream flows
NIC -->|Raw Packets| PCAP
PCAP -->|Packet Data| NMS
CFG -->|Settings| NMS
USR -->|Commands| GUI
GUI -->|Control| NMS

%% Downstream flows
NMS -->|Statistics| GUI
NMS -->|Packet Data| DB
NMS -->|Log Messages| LOG
GUI -->|Export Data| EXP
API -->|gRPC| EXT

%% Dependencies
```

```
NMS -.->|Uses| PCAP
GUI -.->|Uses| QT
NMS -.->|Uses| SSL
```

Data Flow Details

Inbound Data Flows

Network Packet Capture
Source: Network Interface Card (NIC)
Protocol: Raw Ethernet frames
Processing: libpcap → NetworkMonitor → Packet parsing
Rate: Real-time, continuous
Format: Binary packet data
Configuration Input
Source: config/default.conf file
Format: INI-style configuration
Loading: At startup via ConfigManager
Parameters: Interface, filters, storage settings
User Commands
Source: Qt6 GUI interactions
Format: Qt signals/slots
Commands: Start/stop monitoring, set filters, export data

Outbound Data Flows

Database Storage
Destination: SQLite database file
Format: Structured packet records
Frequency: Batch inserts every 5 seconds
Data: Packet metadata and payloads
Log Output
Destination: network_monitor.log file
Format: Timestamped text messages
Levels: DEBUG, INFO, WARNING, ERROR, FATAL
GUI Updates
Destination: Qt6 widgets

Format: Statistics objects, packet lists
Frequency: Configurable (default 1 second)
Data Export (Planned)
Destination: CSV/JSON files
Format: Structured packet data
Trigger: User-initiated
API Access (Planned)
Destination: External systems via gRPC
Format: Protocol Buffer messages
Port: 8080 (configurable)

Integration Points

Message Queue Integration (Not Implemented)

No message brokers (Kafka, RabbitMQ, MQTT) detected in current implementation.

Event Bus (Internal Only)

Qt's signal/slot mechanism for GUI events
Condition variables for thread synchronization

Database as Integration Point

SQLite database could be accessed by external tools
No database replication or synchronization implemented

Integration Risks and Vulnerabilities

Security Risks
Requires root/admin privileges for packet capture
No authentication on planned gRPC API
Unencrypted database storage
No packet data sanitization
Performance Risks
Single SQLite database (potential bottleneck)
No connection pooling
Unbounded packet queue (memory risk)

Reliability Risks

No error recovery for database failures

No reconnection logic for network interfaces

Single point of failure (monolithic architecture)

Compatibility Risks

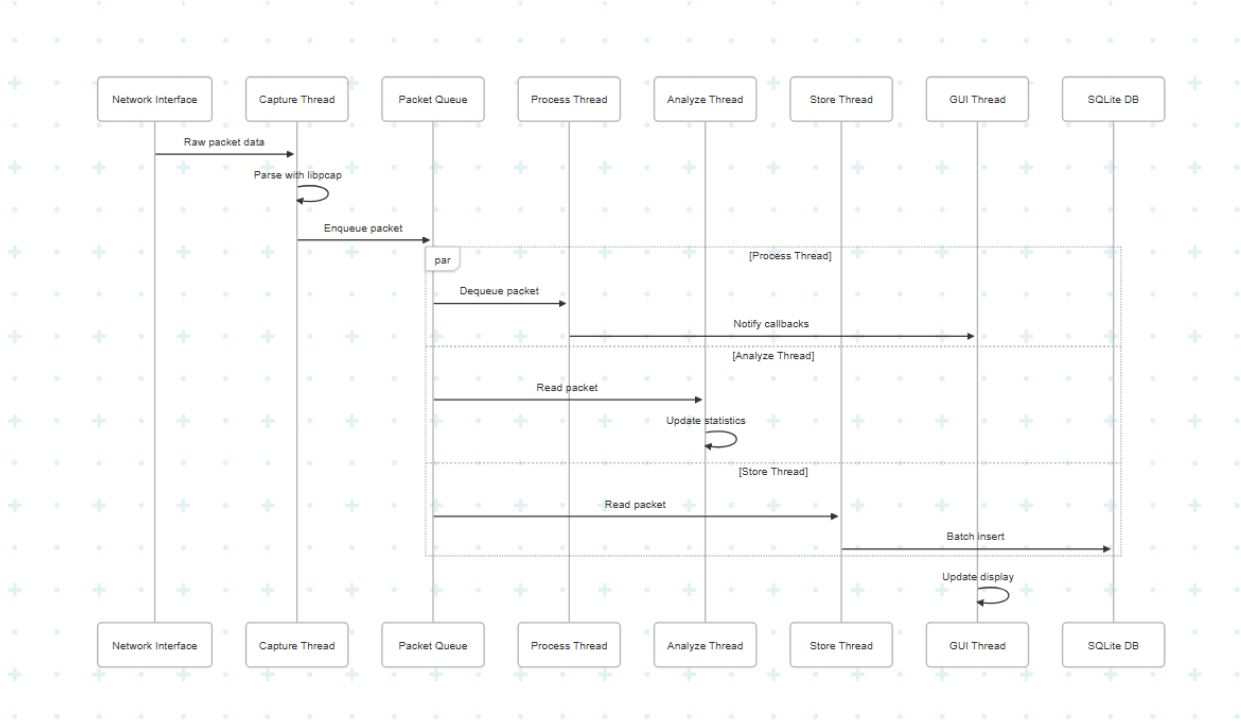
libpcap version dependencies

Qt6 framework requirements

Platform-specific network access

Data Flow Architecture

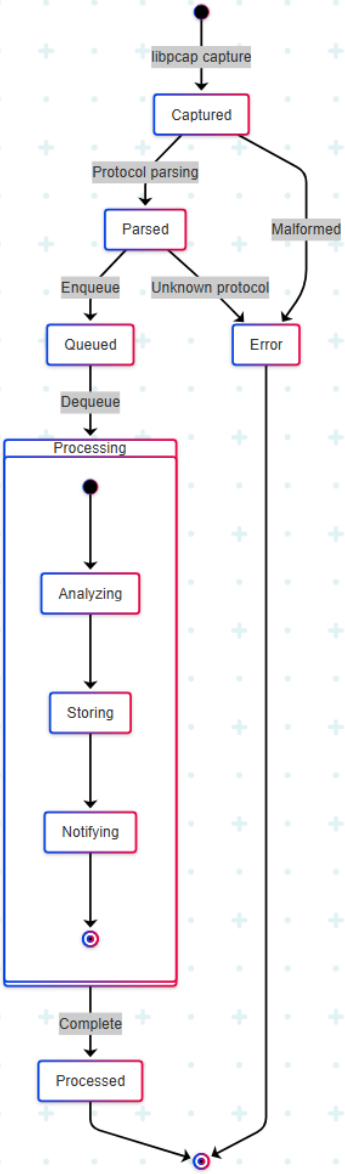
Multi-threaded Processing Pipeline




```
sequenceDiagram
    participant NIC as Network Interface
    participant CT as Capture Thread
    participant PQ as Packet Queue
    participant PT as Process Thread
    participant AT as Analyze Thread
    participant ST as Store Thread
    participant GUI as GUI Thread
```

```
participant DB as SQLite DB

NIC->>CT: Raw packet data
CT->>CT: Parse with libpcap
CT->>PQ: Enqueue packet

par Process Thread
    PQ->>PT: Dequeue packet
    PT->>GUI: Notify callbacks
and Analyze Thread
    PQ->>AT: Read packet
    AT->>AT: Update statistics
and Store Thread
    PQ->>ST: Read packet
    ST->>DB: Batch insert
end

GUI->>GUI: Update display
```

Packet Processing State Machine




```
stateDiagram-v2
    [*] --> Captured: libpcap capture
    Captured --> Parsed: Protocol parsing
    Parsed --> Queued: Enqueue
    Queued --> Processing: Dequeue

    state Processing {
        [*] --> Analyzing
```

```
        Analyzing --> Storing
        Storing --> Notifying
        Notifying --> [*]
    }

    Processing --> Processed: Complete
    Processed --> [*]

    Captured --> Error: Malformed
    Parsed --> Error: Unknown protocol
    Error --> [*]
```

User Stories

Epic 1: Network Traffic Monitoring

User Story 1.1: Start Network Monitoring

As a network administrator

I want to start monitoring network traffic on a specific interface

So that I can analyze network activity in real-time

Acceptance Criteria:

- User can select network interface from dropdown list
- User can click "Start Monitoring" button
- System begins capturing packets on selected interface
- Status bar shows "Monitoring active" with packet count
- System requires elevated privileges (prompts if needed)

Technical Details:

- API: `NetworkMonitor::setInterface(string)`
- API: `NetworkMonitor::start()`
- Error handling for permission denied
- Interface validation before starting

Test Scenarios:

- Start monitoring with valid interface
- Start monitoring without privileges (expect error)
- Start monitoring on non-existent interface
- Start monitoring when already running

User Story 1.2: Apply Packet Filters

As a network analyst

I want to filter captured packets using BPF expressions

So that I can focus on specific traffic patterns

Acceptance Criteria:

- User can open filter dialog from menu/toolbar
- User can enter BPF filter expression
- System validates filter syntax before applying
- Active filter shown in status bar
- User can clear filter to see all traffic

Technical Details:

- API: `NetworkMonitor::setFilter(string)`
- BPF syntax validation via pcap_compile
- Filter persistence in QSettings
- Real-time filter application

Test Scenarios:

- Apply valid BPF filter (e.g., "tcp port 80")
- Apply invalid filter syntax (expect error)
- Clear active filter
- Change filter while monitoring active

Epic 2: Traffic Analysis and Statistics

User Story 2.1: View Protocol Distribution

As a security analyst

I want to see the distribution of network protocols

So that I can identify unusual traffic patterns

Acceptance Criteria:

- Statistics tab shows protocol breakdown
- Pie chart displays protocol percentages
- Table shows packet and byte counts per protocol
- Data updates in real-time (1-second intervals)
- User can sort table by any column

Technical Details:

- API: `Statistics::getTopProtocols(size_t)`
- Qt Charts for visualization
- QTableWidget for detailed view
- Automatic refresh via QTimer

Data Model:

```
struct ProtocolStats {
    Protocol protocol;
    uint64_t packet_count;
    uint64_t byte_count;
    double percentage;
}
```

User Story 2.2: Monitor Bandwidth Usage

As a network administrator

I want to monitor real-time bandwidth usage

So that I can detect network congestion

Acceptance Criteria:

- Bandwidth widget shows current throughput
- Line graph displays bandwidth history (1 hour)
- Shows separate upload/download rates
- Configurable measurement units (Mbps/MBps)
- Export bandwidth data to CSV

Technical Details:

- API: `Statistics::getCurrentBandwidth()`
- API: `Statistics::getBandwidthHistory()`
- QChart with time series data
- Sliding window for history

Epic 3: Connection Tracking

User Story 3.1: View Active Connections

As a security analyst

I want to see all active network connections

So that I can identify suspicious connections

Acceptance Criteria:

- Connections tab lists all active TCP/UDP flows
- Shows source/destination IP and ports
- Displays connection duration and data volume
- Color coding for connection states
- Filter connections by host or port

Technical Details:

- Connection ID: "src_ip:src_port-dst_ip:dst_port"
- API: `Statistics::getActiveConnections()`
- Connection timeout: 5 minutes idle
- State tracking for TCP connections

Data Model:

```
struct Connection {
    string source_address;
    uint16_t source_port;
    string destination_address;
    uint16_t destination_port;
    Protocol protocol;
    ConnectionState state;
    uint64_t bytes_sent;
    uint64_t bytes_received;
    time_point start_time;
    time_point last_activity;
}
```

Epic 4: Packet Inspection

User Story 4.1: View Packet Details

As a network engineer

I want to inspect individual packet contents

So that I can troubleshoot network issues

Acceptance Criteria:

- Packets tab shows real-time packet list
- Double-click packet for detailed view
- Shows all protocol headers (Ethernet, IP, TCP/UDP)
- Hex dump of packet payload
- ASCII representation where applicable

Technical Details:

- API: `Packet` class with all fields
- QTreeWidget for protocol hierarchy
- QHexEdit widget for payload view
- Protocol-specific parsing

User Story 4.2: Search Packets

As a security analyst

I want to search through captured packets

So that I can find specific patterns or content

Acceptance Criteria:

- Search bar in packets tab
- Search by IP address, port, or protocol
- Full-text search in packet payloads
- Highlight matching packets
- Export search results

Technical Details:

- API: `DataStore::getPacketsByHost()`
- API: `DataStore::getPacketsByProtocol()`
- SQL LIKE queries for payload search
- Result pagination for performance

Epic 5: Data Persistence and Export

User Story 5.1: Export Captured Data

As a network analyst

I want to export captured packets to standard formats

So that I can analyze data in other tools

Acceptance Criteria:

- Export menu with format options
- Support PCAP format for Wireshark
- Support CSV for statistics
- Support JSON for structured data
- Progress dialog for large exports

Technical Details:

- libpcap API for PCAP export
- Custom CSV/JSON formatters
- Async export with progress callback
- File size warnings for large datasets

Export Formats:

```
{
  "export_info": {
    "timestamp": "2024-01-01T00:00:00Z",
    "interface": "eth0",
    "filter": "tcp port 80",
    "packet_count": 1000
  },
  "packets": [
    {
      "timestamp": "2024-01-01T00:00:01.123Z",
      "protocol": "TCP",
      "source": "192.168.1.100:54321",
      "destination": "93.184.216.34:80",
      "length": 1500,
      "flags": ["ACK", "PSH"]
    }
  ]
}
```

User Story 5.2: Manage Storage Limits

As a system administrator

I want to configure storage limits for packet data

So that disk space is not exhausted

Acceptance Criteria:

- Settings dialog with storage options
- Maximum database size configuration

Automatic cleanup of old packets
Warning when approaching limit
Manual purge option

Technical Details:

Config: max_packets = 1000000
Background cleanup thread
FIFO deletion policy
Database VACUUM after cleanup

Epic 6: System Configuration

User Story 6.1: Configure Application Settings

As a user

I want to customize application settings

So that the tool works according to my preferences

Acceptance Criteria:

Settings dialog accessible from menu
General settings (theme, language)
Monitoring settings (buffer size, timeout)
Storage settings (location, limits)
Settings persist between sessions

Technical Details:

QSettings for persistence
Config file: default.conf
Hot-reload for some settings
Validation for numeric inputs

Settings Structure:

```
[general]
theme = dark
language = en_US
```

```
[monitoring]
buffer_size = 65536
timeout = 1000

[storage]
database_path = ./network_monitor.db
max_size_mb = 1000
```

Epic 7: API Integration (Future)

User Story 7.1: Enable API Access

As a DevOps engineer

I want to access monitoring data via API

So that I can integrate with other systems

Acceptance Criteria:

- gRPC API endpoint on configurable port
- Protocol Buffer message definitions
- Authentication via API keys
- Rate limiting to prevent abuse
- API documentation

Technical Details:

- gRPC service definition
- TLS encryption for API
- JWT token authentication
- Prometheus metrics endpoint

API Example:

```
service NetworkMonitor {
  rpc GetStatistics(Empty) returns (Statistics);
  rpc GetPackets(PacketFilter) returns (stream Packet);
  rpc GetConnections(Empty) returns (ConnectionList);
```

```
}
```

Jira Export Format

```
Summary,Issue Type,Description,Acceptance Criteria,Story Points,Epic Link,Components,Labels
"Start Network Monitoring",Story,"As a network administrator I want to start monitoring network traffic on a specific interface so that I can analyze network activity in real-time","1. User can select network interface from dropdown list
2. User can click Start Monitoring button
3. System begins capturing packets on selected interface
4. Status bar shows Monitoring active with packet count
5. System requires elevated privileges",5,Network Traffic Monitoring,Core,monitoring
"Apply Packet Filters",Story,"As a network analyst I want to filter captured packets using BPF expressions so that I can focus on specific traffic patterns","1. User can open filter dialog from menu/toolbar
2. User can enter BPF filter expression
3. System validates filter syntax before applying
4. Active filter shown in status bar
5. User can clear filter to see all traffic",3,Network Traffic Monitoring,Core,filtering
"View Protocol Distribution",Story,"As a security analyst I want to see the distribution of network protocols so that I can identify unusual traffic patterns","1. Statistics tab shows protocol breakdown
2. Pie chart displays protocol percentages
3. Table shows packet and byte counts per protocol
4. Data updates in real-time
5. User can sort table by any column",3,Traffic Analysis,GUI,analytics
```

Conclusion

This Network Monitoring System represents a comprehensive solution for real-time network traffic analysis. The architecture leverages modern C++ features, established libraries like libpcap and Qt6, and follows solid design principles with clear separation of concerns.

Key Architectural Decisions

Multi-threaded design for performance and responsiveness
SQLite for lightweight, embedded data persistence
Qt6 for cross-platform GUI development
libpcap for reliable packet capture
Plugin architecture for extensibility

Areas for Enhancement

Implement the planned gRPC API for external integrations
Add support for more protocols (IPv6, QUIC, etc.)
Implement data encryption for sensitive packet storage
Add machine learning capabilities for anomaly detection
Implement distributed architecture for scalability

The system provides a solid foundation for network monitoring while maintaining flexibility for future enhancements and integrations.