Network Monitoring System - Comprehensive Project Documentation

Table of Contents

Executive Summary

The Network Monitoring System is a comprehensive C++20-based application designed for real-time network traffic analysis and monitoring. The system features a Qt6-based GUI, multi-threaded packet capture using libpcap, SQLite-based data persistence, and extensive protocol analysis capabilities.

Key Features

- Real-time packet capture and analysis using libpcap
- Support for multiple protocols (TCP, UDP, ICMP, HTTP, HTTPS, DNS, DHCP, ARP)
- Qt6-based graphical user interface with multiple visualization widgets
- SQLite database for packet storage and historical analysis
- Multi-threaded architecture for optimal performance
- Configurable filtering using Berkeley Packet Filter (BPF) expressions
- Both CLI and GUI interfaces

- Extensible plugin system architecture

Technology Stack

- **Language**: C++20
- **GUI Framework**: Qt6 (Core, Gui, Widgets, Charts)
- **Packet Capture**: libpcap
- **Database**: SQLite3
- **Networking**: Boost (system, filesystem)
- **Security**: OpenSSL
- **API**: gRPC with Protocol Buffers
- **Build System**: CMake 3.15+

Project Overview

Project Type

Network monitoring system with C++ backend and Qt6 frontend

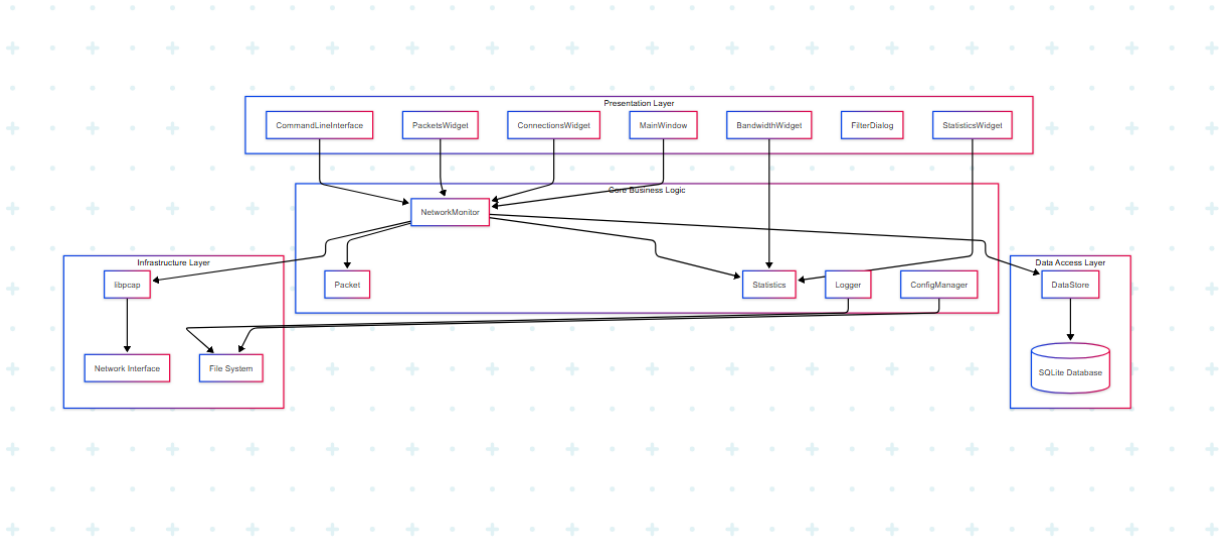
Business Domain

Real-time network traffic monitoring, protocol analysis, bandwidth monitoring, and security analysis

Stakeholders

- Network administrators
- Security analysts
- DevOps teams
- System administrators
- Development team (backend and frontend)

Current State Architecture




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

    subgraph "Core Business Logic"
        NM[NetworkMonitor]
        PKT[Packet]
        STAT[Statistics]
        CFG[ConfigManager]
        LOG[Logger]
    end

    subgraph "Data Access Layer"
        DS[DataStore]
        DB[(SQLite Database)]
    end
```

```
subgraph "Infrastructure Layer"
    PCAP[libpcap]
    NET[Network Interface]
    FS[File System]
end

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

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

DS --> DB
PCAP --> NET
CFG --> FS
LOG --> FS
```

Architecture Patterns

- **Multi-threaded Event-Driven Architecture**: Separate threads for capture, processing, analysis, and storage
- **Model-View-Controller (MVC)**: GUI components follow MVC pattern
- **Observer Pattern**: GUI widgets observe NetworkMonitor state changes
- **Singleton Pattern**: Logger and ConfigManager use singleton pattern
- **Factory Pattern**: Packet parsing uses factory pattern for different protocols

File-by-File Tree Structure

Root Directory Structure

```
network-monitoring/
├── src/                    # Source code directory
├── include/                # Header files directory
├── config/                 # Configuration files
├── Documentation/          # Project documentation
├── CMakeLists.txt          # Main CMake build configuration
├── README.md               # Project overview and setup instructions
├── .gitignore              # Git ignore patterns
└── .git/                   # Git repository metadata
```

Detailed File Structure

/src/ - Source Files

- **Purpose**: Contains all C++ implementation files
- **Contents**: Core functionality, GUI components, utilities
- **Dependencies**: Qt6, libpcap, SQLite3, Boost

```
src/
├── main.cpp                    # Application entry point
├── CMakeLists.txt              # Source-level CMake configuration
├── core/
│   └── NetworkMonitor.cpp      # Core monitoring functionality
├── protocols/
│   └── Packet.cpp              # Packet parsing and analysis
├── analysis/
│   └── Statistics.cpp          # Statistical analysis engine
├── storage/
│   └── DataStore.cpp           # SQLite database operations
├── utils/
│   └── Logger.cpp              # Logging system implementation
├── config/
│   └── ConfigManager.cpp       # Configuration management
├── gui/
```

```
│   ├── MainWindow.cpp         # Main application window
│   ├── StatisticsWidget.cpp   # Statistics display widget
│   ├── ConnectionsWidget.cpp  # Connections display widget
│   ├── PacketsWidget.cpp      # Packet list widget
│   ├── BandwidthWidget.cpp    # Bandwidth chart widget
│   └── FilterDialog.cpp       # Filter configuration dialog
└── cli/
    └── CommandLineInterface.cpp # CLI implementation
```

/include/ - Header Files

- **Purpose**: Contains all C++ header files with class declarations
- **Contents**: Interface definitions, class declarations, constants

```
include/
├── core/
│   └── NetworkMonitor.hpp     # NetworkMonitor class interface
├── protocols/
│   └── Packet.hpp             # Packet structure and parsing
interface
├── analysis/
│   └── Statistics.hpp         # Statistics analysis interface
├── storage/
│   └── DataStore.hpp          # Database storage interface
├── utils/
│   └── Logger.hpp             # Logging system interface
├── config/
│   └── ConfigManager.hpp      # Configuration management interface
├── gui/
│   ├── MainWindow.hpp         # Main window class interface
│   ├── StatisticsWidget.hpp   # Statistics widget interface
│   ├── ConnectionsWidget.hpp  # Connections widget interface
│   ├── PacketsWidget.hpp      # Packets widget interface
│   ├── BandwidthWidget.hpp    # Bandwidth widget interface
│   └── FilterDialog.hpp       # Filter dialog interface
└── cli/
    └── CommandLineInterface.hpp # CLI interface
```

/config/ - Configuration Files

```
config/
└── default.conf                # Default application configuration
```

Sitemap

Core Components Sitemap

Component	File Path	Purpose	Dependencies	User Stories
NetworkMonitor	`src/core/NetworkMonitor.cpp`	Core packet capture and monitoring	libpcap, Statistics, DataStore	NET-001, NET-002
Packet	`src/protocols/Packet.cpp`	Packet parsing and protocol analysis	Standard library	NET-003, NET-004
Statistics	`src/analysis/Statistics.cpp`	Real-time statistical analysis	Packet	NET-005, NET-006
DataStore	`src/storage/DataStore.cpp`	SQLite database operations	SQLite3, Packet	NET-007, NET-008
Logger	`src/utils/Logger.cpp`	Application logging system	File system	NET-009
ConfigManager	`src/config/ConfigManager.cpp`	Configuration management	File system	NET-010

GUI Components Sitemap

Component	File Path	Purpose	Dependencies	User Stories

MainWindow	`src/gui/MainWindow.cpp`	Main application window	Qt6, NetworkMonitor	GUI-001, GUI-002
StatisticsWidget	`src/gui/StatisticsWidget.cpp`	Statistics display	Qt6, Statistics	GUI-003
ConnectionsWidget	`src/gui/ConnectionsWidget.cpp`	Active connections display	Qt6, NetworkMonitor	GUI-004
PacketsWidget	`src/gui/PacketsWidget.cpp`	Packet list display	Qt6, Packet	GUI-005
BandwidthWidget	`src/gui/BandwidthWidget.cpp`	Bandwidth visualization	Qt6 Charts, Statistics	GUI-006
FilterDialog	`src/gui/FilterDialog.cpp`	Filter configuration	Qt6	GUI-007

CLI Components Sitemap

Component	File Path	Purpose	Dependencies	User Stories
CommandLineInterface	`src/cli/CommandLineInterface.cpp`	Command-line interface	NetworkMonitor	CLI-001, CLI-002

API Documentation

Core API Interfaces

NetworkMonitor Class

```
class NetworkMonitor {
public:
    void start();                    // Start packet capture
    void stop();                     // Stop packet capture
```

```
    void setInterface(const std::string& interface);  // Set network
interface
    void setFilter(const std::string& filter);        // Set BPF
filter
    Statistics getStatistics() const;                 // Get current
statistics
    void addPacketCallback(std::function<void(const Packet&)>
callback);
};
```

Methods:

- `start()`: Initializes packet capture threads and begins monitoring
- `stop()`: Gracefully stops all monitoring threads
- `setInterface(interface)`: Configures network interface for monitoring
- `setFilter(filter)`: Sets Berkeley Packet Filter expression
- `getStatistics()`: Returns current network statistics
- `addPacketCallback(callback)`: Registers packet processing callback

Statistics Class

```
class Statistics {
public:
    void update(const Packet& packet);               // Update with
new packet
    void reset();                                    // Reset all
statistics
    uint64_t getTotalPackets() const;               // Get total
packet count
    uint64_t getTotalBytes() const;                 // Get total byte
count
    double getCurrentBandwidth() const;             // Get current
bandwidth
    std::vector<std::pair<Packet::Protocol, uint64_t>>
getTopProtocols(size_t count) const;
    std::vector<std::pair<std::string, uint64_t>> getTopHosts(size_t
count) const;
```

```
};
```

DataStore Class

```
class DataStore {
public:
    void store(const Packet& packet);               // Store packet to
database
    void flush();                                   // Flush pending
writes
    std::vector<Packet> getPacketsByProtocol(Packet::Protocol
protocol, size_t limit);
    std::vector<Packet> getPacketsByHost(const std::string& host,
size_t limit);
    std::vector<Packet> getPacketsByTimeRange(const
std::chrono::system_clock::time_point& start,
                                              const
std::chrono::system_clock::time_point& end,
                                              size_t limit);
};
```

Function Design Documentation

Core Functions

NetworkMonitor::start()

- **Purpose**: Initialize and start packet capture
- **Input**: None
- **Output**: void
- **Logic**:
    - Initialize libpcap handle
    - Set promiscuous mode
    - Apply BPF filter if configured
    - Start capture, process, analyze, and store threads
- **File Reference**: `src/core/NetworkMonitor.cpp:45-78`

Packet::parseEthernet()

- **Purpose**: Parse Ethernet frame header
- **Input**: Raw packet data
- **Output**: void (updates packet fields)
- **Logic**:
  - Extract destination MAC address
  - Extract source MAC address
  - Extract EtherType
  - Determine next protocol layer
- **File Reference**: `src/protocols/Packet.cpp:89-112`

Statistics::update()

- **Purpose**: Update statistics with new packet data
- **Input**: const Packet& packet
- **Output**: void
- **Logic**:
  - Update total counters
  - Update protocol-specific statistics
  - Update host statistics
  - Update bandwidth calculations
  - Update connection tracking
- **File Reference**: `src/analysis/Statistics.cpp:67-89`

GUI Functions

MainWindow::updateDisplay()

- **Purpose**: Refresh all GUI widgets with current data
- **Input**: None (slot function)
- **Output**: void
- **Logic**:
  - Get current statistics from NetworkMonitor
  - Update each widget with new data
  - Refresh charts and tables
- **File Reference**: `src/gui/MainWindow.cpp:156-178`

Narrative Descriptions

Core Flow: Packet Capture and Analysis

The network monitoring system follows a multi-threaded pipeline architecture for packet processing:

1. **Packet Capture Thread** (`NetworkMonitor::captureThread()`):
    a. Continuously captures packets from the network interface using libpcap
    b. Applies BPF filters to reduce processing overhead
    c. Queues captured packets for processing
    d. Located in: `src/core/NetworkMonitor.cpp`
2. **Packet Processing Thread** (`NetworkMonitor::processThread()`):
    a. Dequeues raw packets from capture queue
    b. Parses packet headers (Ethernet, IP, TCP/UDP)
    c. Determines protocol types and extracts metadata
    d. Creates Packet objects with parsed information
    e. Located in: `src/protocols/Packet.cpp`
3. **Analysis Thread** (`NetworkMonitor::analyzeThread()`):
    a. Receives parsed packets for statistical analysis
    b. Updates real-time statistics (bandwidth, protocol distribution)
    c. Tracks active connections and hosts
    d. Maintains historical data for trending
    e. Located in: `src/analysis/Statistics.cpp`
4. **Storage Thread** (`NetworkMonitor::storeThread()`):
    a. Persists packet data to SQLite database
    b. Implements batch insertion for performance
    c. Manages database cleanup and maintenance
    d. Located in: `src/storage/DataStore.cpp`

GUI Flow: Real-time Visualization

The Qt6-based GUI provides real-time visualization through a timer-based update mechanism:

1. **Main Window Initialization** (`MainWindow::MainWindow()`):
    a. Creates tabbed interface with specialized widgets
    b. Establishes connection to NetworkMonitor

    c. Sets up periodic update timer (1-second intervals)
    d. Located in: `src/gui/MainWindow.cpp`
2. **Widget Update Cycle** (`MainWindow::updateDisplay()`):
    a. Triggered by QTimer every second
    b. Retrieves current statistics from NetworkMonitor
    c. Updates each widget with fresh data
    d. Refreshes charts, tables, and labels
3. **Specialized Widget Functions**:
    a. **StatisticsWidget**: Displays protocol distribution and totals
    b. **ConnectionsWidget**: Shows active network connections
    c. **PacketsWidget**: Lists recent packets with details
    d. **BandwidthWidget**: Renders real-time bandwidth charts

Configuration Flow

The system uses a hierarchical configuration approach:

1. **Default Configuration** (`config/default.conf`):
    a. Provides baseline settings for all components
    b. Includes monitoring parameters, storage settings, GUI preferences
2. **Runtime Configuration** (`ConfigManager`):
    a. Loads configuration from files and command-line arguments
    b. Provides type-safe access to configuration values
    c. Supports dynamic configuration updates

Dependencies and External Systems

External Libraries

- **libpcap**: Packet capture library for network monitoring
- **SQLite3**: Embedded database for packet storage
- **Qt6**: Cross-platform GUI framework
- **Boost**: C++ utility libraries (system, filesystem)
- **OpenSSL**: Cryptographic library for secure communications
- **gRPC**: High-performance RPC framework
- **Protocol Buffers**: Data serialization library

System Dependencies

- **Network Interface**: Physical or virtual network adapter
- **File System**: For configuration, logs, and database storage
- **Operating System**: Linux/Windows/macOS support through Qt6 and libpcap

Security Analysis

Identified Security Considerations

1. **Packet Capture Privileges**: Requires elevated privileges for promiscuous mode
2. **Data Storage**: Sensitive network data stored in SQLite database
3. **Memory Management**: C++ manual memory management for packet buffers
4. **Input Validation**: BPF filter expressions need validation

Security Measures

1. **Privilege Dropping**: Application should drop privileges after initialization
2. **Data Encryption**: Consider encrypting stored packet data
3. **Access Control**: Implement user authentication for GUI access
4. **Audit Logging**: Comprehensive logging of all operations

Performance Analysis

Performance Characteristics

1. **Multi-threading**: Separate threads for capture, processing, analysis, storage
2. **Queue-based Architecture**: Asynchronous processing with bounded queues
3. **Batch Processing**: Database operations use batch insertion
4. **Memory Management**: Efficient packet buffer management

Potential Bottlenecks

1. **Packet Capture Rate**: Limited by network interface and libpcap performance
2. **Database I/O**: SQLite write performance under high packet rates
3. **GUI Updates**: Qt widget refresh rate may impact performance
4. **Memory Usage**: Large packet queues may consume significant memory

Testing and Quality

Testing Strategy

1. **Unit Tests**: Individual component testing (recommended: Google Test)
2. **Integration Tests**: End-to-end packet processing pipeline
3. **Performance Tests**: Load testing with high packet rates
4. **GUI Tests**: Qt Test framework for widget functionality

Code Quality Metrics

1. **Cyclomatic Complexity**: Monitor function complexity
2. **Code Coverage**: Aim for >80% test coverage
3. **Static Analysis**: Use tools like Cppcheck, clang-static-analyzer
4. **Memory Leak Detection**: Valgrind for memory leak detection

Generated on: $(date) Project Version: 1.0.0 Documentation Version: 1.0