Network Monitoring System - API Documentation

Table of Contents

Overview

The Network Monitoring System provides several API layers for different types of interactions:

- **Core C++ APIs**: Internal class interfaces for packet capture, analysis, and storage
- **GUI APIs**: Qt6-based interfaces for user interaction
- **Configuration APIs**: System configuration and management
- **Future gRPC APIs**: External integration capabilities (planned)

Core API Classes

NetworkMonitor Class API

File: `include/core/NetworkMonitor.hpp`, `src/core/NetworkMonitor.cpp`

Constructor/Destructor

```
NetworkMonitor();
~NetworkMonitor();
```

Public Methods

start()

```
void start();
```

- **Purpose**: Initialize and start packet capture
- **Parameters**: None
- **Returns**: void
- **Throws**: `std::runtime_error` if interface initialization fails
- **Thread Safety**: Thread-safe
- **Example**:
```
NetworkMonitor monitor;
monitor.setInterface("eth0");
monitor.start();
```

stop()

```
void stop();
```

- **Purpose**: Stop packet capture and cleanup resources
- **Parameters**: None
- **Returns**: void
- **Thread Safety**: Thread-safe
- **Side Effects**: Stops all capture threads, flushes data to storage

setInterface()

```
void setInterface(const std::string& interface);
```

- **Purpose**: Set network interface for packet capture

- **Parameters**:
  - `interface`: Network interface name (e.g., "eth0", "wlan0")
- **Returns**: void
- **Throws**: `std::invalid_argument` if interface doesn't exist
- **Preconditions**: Must be called before `start()`

setFilter()

```
void setFilter(const std::string& filter);
```

- **Purpose**: Set Berkeley Packet Filter (BPF) expression
- **Parameters**:
  - `filter`: BPF filter string (e.g., "tcp port 80", "host 192.168.1.1")
- **Returns**: void
- **Throws**: `std::invalid_argument` if filter syntax is invalid
- **Example**:`monitor.setFilter("tcp and port 443");`

getStatistics()

```
Statistics getStatistics() const;
```

- **Purpose**: Get current network statistics
- **Parameters**: None
- **Returns**: `Statistics` object containing current metrics
- **Thread Safety**: Thread-safe (returns copy)

addPacketCallback()

```
void addPacketCallback(std::function<void(const Packet&)> callback);
```

- **Purpose**: Register callback for packet processing
- **Parameters**:
  - `callback`: Function to call for each captured packet
- **Returns**: void
- **Thread Safety**: Thread-safe

- **Performance**: Callbacks are executed in capture thread context

Packet Structure API

File: `include/protocols/Packet.hpp`, `src/protocols/Packet.cpp`

Constructor

```
Packet(const uint8_t* data, size_t length, const struct timeval&
timestamp);
```

Public Members

```
struct Packet {
    enum class Protocol {
        UNKNOWN, ETHERNET, IPV4, IPV6, TCP, UDP, ICMP,
        HTTP, HTTPS, DNS, DHCP, ARP
    };

    std::vector<uint8_t> raw_data;              // Raw packet data
    size_t length;                              // Packet length
    std::chrono::system_clock::time_point timestamp; // Capture
timestamp
    Protocol protocol;                          // Detected protocol
    std::string source_address;                 // Source IP address
    std::string destination_address;            // Destination IP
address
    uint16_t source_port;                       // Source port (if
applicable)
    uint16_t destination_port;                  // Destination port
(if applicable)
    bool is_fragmented;                         // IP fragmentation
flag
    bool is_malformed;                          // Malformed packet
flag
    std::vector<uint8_t> payload;               // Application payload
};
```

Public Methods

Protocol Detection Methods

```
bool isTCP() const;
bool isUDP() const;
bool isICMP() const;
bool isHTTP() const;
bool isHTTPS() const;
bool isDNS() const;
bool isARP() const;
bool isIPv4() const;
bool isIPv6() const;
```

- **Purpose**: Check if packet matches specific protocol
- **Returns**: `true` if packet is of specified protocol
- **Thread Safety**: Thread-safe (const methods)

getProtocolString()

```
std::string getProtocolString() const;
```

- **Purpose**: Get human-readable protocol name
- **Returns**: Protocol name as string (e.g., "TCP", "UDP", "HTTP")

Statistics Class API

File: `include/analysis/Statistics.hpp`, `src/analysis/Statistics.cpp`

Public Methods

update()

```
void update(const Packet& packet);
```

- **Purpose**: Update statistics with new packet data
- **Parameters**:

    - `packet`: Packet to analyze and add to statistics
- **Returns**: void
- **Thread Safety**: Thread-safe (uses internal locking)
- **Performance**: O(1) for most operations

reset()

```
void reset();
```

- **Purpose**: Reset all statistics to zero
- **Thread Safety**: Thread-safe

Basic Statistics

```
uint64_t getTotalPackets() const;
uint64_t getTotalBytes() const;
uint64_t getErrorCount() const;
```

- **Purpose**: Get basic traffic counters
- **Returns**: Current count values
- **Thread Safety**: Thread-safe (atomic operations)

Protocol Statistics

```
uint64_t getProtocolPacketCount(Packet::Protocol protocol) const;
uint64_t getProtocolByteCount(Packet::Protocol protocol) const;
std::vector<std::pair<Packet::Protocol, uint64_t>>
getTopProtocols(size_t count) const;
```

- **Purpose**: Get protocol-specific statistics
- **Parameters**:
    - `protocol`: Specific protocol to query
    - `count`: Number of top protocols to return
- **Returns**: Protocol statistics or top protocols list

Host Statistics

```
std::vector<std::pair<std::string, uint64_t>> getTopHosts(size_t
count) const;
HostStats getHostStats(const std::string& host) const;
std::vector<std::string> getActiveHosts() const;
```

- **Purpose**: Get host-based traffic statistics
- **Returns**: Host statistics and active host lists

Bandwidth Statistics

```
double getCurrentBandwidth() const;
double getAverageBandwidth() const;
std::vector<std::pair<std::chrono::system_clock::time_point, double>>
getBandwidthHistory() const;
```

- **Purpose**: Get bandwidth utilization metrics
- **Returns**: Bandwidth in bits per second

DataStore Class API

File: `include/storage/DataStore.hpp`, `src/storage/DataStore.cpp`

Constructor

```
DataStore(const std::string& db_path = "network_monitor.db");
```

Storage Methods

store()

```
void store(const Packet& packet);
```

- **Purpose**: Store packet to database
- **Parameters**:
  - `packet`: Packet to store

- **Returns**: void
- **Thread Safety**: Thread-safe (uses internal queue)
- **Performance**: Asynchronous operation with batching

flush()

```
void flush();
```

- **Purpose**: Force flush of pending writes to database
- **Returns**: void
- **Blocking**: May block until all pending writes complete

Query Methods

getPacketsByProtocol()

```
std::vector<Packet> getPacketsByProtocol(Packet::Protocol protocol,
size_t limit = 1000);
```

- **Purpose**: Retrieve packets by protocol type
- **Parameters**:
  - `protocol`: Protocol to filter by
  - `limit`: Maximum number of packets to return
- **Returns**: Vector of matching packets
- **Performance**: Indexed query, O(log n)

getPacketsByHost()

```
std::vector<Packet> getPacketsByHost(const std::string& host, size_t
limit = 1000);
```

- **Purpose**: Retrieve packets by host address
- **Parameters**:
  - `host`: IP address to filter by
  - `limit`: Maximum number of packets to return
- **Returns**: Vector of matching packets

getPacketsByTimeRange()

```
std::vector<Packet> getPacketsByTimeRange(
    const std::chrono::system_clock::time_point& start,
    const std::chrono::system_clock::time_point& end,
    size_t limit = 1000
);
```

- **Purpose**: Retrieve packets within time range
- **Parameters**:
  - `start`: Start time for query
  - end: End time for query
  - `limit`: Maximum number of packets to return
- **Returns**: Vector of packets in time range

Statistics Query Methods

getPacketCount()

```
uint64_t getPacketCount();
```

- **Purpose**: Get total stored packet count
- **Returns**: Number of packets in database

getProtocolDistribution()

```
std::vector<std::pair<Packet::Protocol, uint64_t>>
getProtocolDistribution();
```

- **Purpose**: Get protocol distribution from stored data
- **Returns**: Vector of protocol counts

GUI API Interfaces

MainWindow Class API

File: `include/gui/MainWindow.hpp`, `src/gui/MainWindow.cpp`

Constructor

```
MainWindow(NetworkMonitor* monitor, QWidget* parent = nullptr);
```

Public Slots

```
public slots:
    void updateDisplay();                    // Update all widgets
    void showFilterDialog();                 // Show filter
configuration
    void clearFilter();                      // Clear current filter
    void saveStatistics();                   // Save statistics to
file
    void exportData();                       // Export data to file
    void toggleMonitoring(bool start);       // Start/stop monitoring
```

Widget APIs

StatisticsWidget

```
class StatisticsWidget : public QWidget {
public:
    explicit StatisticsWidget(NetworkMonitor* monitor, QWidget* parent
= nullptr);
    void update();                           // Update statistics
display
};
```

ConnectionsWidget

```
class ConnectionsWidget : public QWidget {
public:
    explicit ConnectionsWidget(NetworkMonitor* monitor, QWidget*
parent = nullptr);
    void update();                           // Update connections
display
```

```
};
```

PacketsWidget

```
class PacketsWidget : public QWidget {
public:
    explicit PacketsWidget(NetworkMonitor* monitor, QWidget* parent =
nullptr);
    void update();                          // Update packet list
    void clearPackets();                    // Clear packet display
};
```

BandwidthWidget

```
class BandwidthWidget : public QWidget {
public:
    explicit BandwidthWidget(NetworkMonitor* monitor, QWidget* parent
= nullptr);
    void update();                          // Update bandwidth chart
};
```

Configuration API

ConfigManager Class API

File: `include/config/ConfigManager.hpp`, `src/config/ConfigManager.cpp`

Singleton Access

```
static ConfigManager& getInstance();
```

Configuration Methods

```
std::string getString(const std::string& key, const std::string&
default_value = "");
```

```
int getInt(const std::string& key, int default_value = 0);
bool getBool(const std::string& key, bool default_value = false);
double getDouble(const std::string& key, double default_value = 0.0);

void setString(const std::string& key, const std::string& value);
void setInt(const std::string& key, int value);
void setBool(const std::string& key, bool value);
void setDouble(const std::string& key, double value);

bool loadFromFile(const std::string& filename);
bool saveToFile(const std::string& filename);
```

Configuration Keys

```
// General settings
"general.log_level"          // string: debug, info, warning, error,
fatal
"general.log_file"           // string: log file path
"general.database"           // string: database file path

// Monitoring settings
"monitoring.interface"       // string: network interface name
"monitoring.promiscuous_mode" // bool: enable promiscuous mode
"monitoring.buffer_size"     // int: capture buffer size
"monitoring.timeout"         // int: capture timeout in ms
"monitoring.filter"          // string: BPF filter expression

// Storage settings
"storage.max_packets"        // int: maximum packets to store
"storage.cleanup_interval"   // int: cleanup interval in seconds
"storage.batch_size"         // int: database batch size
"storage.flush_interval"     // int: flush interval in seconds

// GUI settings
"gui.theme"                  // string: light, dark
"gui.refresh_rate"           // int: GUI refresh rate in ms
"gui.max_connections"        // int: max connections to display
```

```
"gui.max_packets_display"    // int: max packets to display
```

Utility APIs

Logger Class API

File: `include/utils/Logger.hpp`, `src/utils/Logger.cpp`

Initialization

```
static void init(const std::string& filename, Level level =
Level::INFO);
```

Logging Methods

```
static void debug(const std::string& message);
static void info(const std::string& message);
static void warning(const std::string& message);
static void error(const std::string& message);
static void fatal(const std::string& message);
```

Log Levels

```
enum class Level {
    DEBUG = 0,
    INFO = 1,
    WARNING = 2,
    ERROR = 3,
    FATAL = 4
};
```

Future gRPC API

Planned Service Definitions

MonitoringService

```
service MonitoringService {
    rpc StartMonitoring(StartRequest) returns (StatusResponse);
    rpc StopMonitoring(StopRequest) returns (StatusResponse);
    rpc GetStatistics(StatisticsRequest) returns (StatisticsResponse);
    rpc SetFilter(FilterRequest) returns (StatusResponse);
    rpc GetPackets(PacketRequest) returns (stream PacketResponse);
}
```

Message Definitions

```
message StartRequest {
    string interface = 1;
    string filter = 2;
}

message StatusResponse {
    bool success = 1;
    string message = 2;
}

message StatisticsResponse {
    uint64 total_packets = 1;
    uint64 total_bytes = 2;
    double current_bandwidth = 3;
    repeated ProtocolStat protocol_stats = 4;
}

message PacketResponse {
    string timestamp = 1;
    string protocol = 2;
    string source_address = 3;
```

```
    string destination_address = 4;
    uint32 source_port = 5;
    uint32 destination_port = 6;
    uint32 length = 7;
}
```

Error Handling

Exception Types

NetworkMonitorException

```
class NetworkMonitorException : public std::runtime_error {
public:
    NetworkMonitorException(const std::string& message);
};
```

Common Error Scenarios

1. **Interface Not Found**
    a. **Exception**: `std::invalid_argument`
    b. **Message**: "Network interface 'eth0' not found"
    c. **Recovery**: Check available interfaces, use valid interface name
2. **Permission Denied**
    a. **Exception**: `std::runtime_error`
    b. **Message**: "Permission denied: packet capture requires root privileges"
    c. **Recovery**: Run with elevated privileges
3. **Invalid Filter**
    a. **Exception**: `std::invalid_argument`
    b. **Message**: "Invalid BPF filter expression: 'invalid syntax'"
    c. **Recovery**: Validate filter syntax, use correct BPF format
4. **Database Error**
    a. **Exception**: `std::runtime_error`
    b. **Message**: "Database error: unable to open database file"
    c. **Recovery**: Check file permissions, disk space

Code Examples

Basic Monitoring Setup

```
#include "core/NetworkMonitor.hpp"
#include "utils/Logger.hpp"

int main() {
    try {
        // Initialize logging
        Logger::init("monitor.log", Logger::Level::INFO);

        // Create and configure monitor
        NetworkMonitor monitor;
        monitor.setInterface("eth0");
        monitor.setFilter("tcp port 80 or tcp port 443");

        // Add packet callback
        monitor.addPacketCallback([](const Packet& packet) {
            Logger::info("Captured packet: " +
packet.getProtocolString());
        });

        // Start monitoring
        monitor.start();

        // Monitor for 60 seconds
        std::this_thread::sleep_for(std::chrono::seconds(60));

        // Get final statistics
        Statistics stats = monitor.getStatistics();
        Logger::info("Total packets: " +
std::to_string(stats.getTotalPackets()));

        // Stop monitoring
        monitor.stop();

    } catch (const std::exception& e) {
```

```
        Logger::error("Error: " + std::string(e.what()));
        return 1;
    }

    return 0;
}
```

GUI Application Setup

```
#include <QApplication>
#include "gui/MainWindow.hpp"
#include "core/NetworkMonitor.hpp"

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);

    // Create monitor
    auto monitor = std::make_unique<NetworkMonitor>();
    monitor->setInterface("eth0");

    // Create and show main window
    MainWindow window(monitor.get());
    window.show();

    return app.exec();
}
```

Database Query Example

```
#include "storage/DataStore.hpp"
#include "protocols/Packet.hpp"

void analyzeTraffic() {
    DataStore store("network_data.db");

    // Get HTTP packets from last hour
    auto now = std::chrono::system_clock::now();
```

```
    auto hour_ago = now - std::chrono::hours(1);

    auto packets = store.getPacketsByTimeRange(hour_ago, now, 1000);

    // Filter for HTTP packets
    for (const auto& packet : packets) {
        if (packet.isHTTP()) {
            std::cout << "HTTP packet: "
                      << packet.source_address << ":" <<
packet.source_port
                      << " -> "
                      << packet.destination_address << ":" <<
packet.destination_port
                      << std::endl;
        }
    }
}
```

Generated on: $(date) API Documentation Version: 1.0 Compatible with: Network Monitor v1.0.0