NMS (Network Monitor System): Computer Network Monitor and Log Collection Platform

Negura Teodor-Alexandru

University “Alexandru Ioan Cuza” of Iași, Faculty of Computer Science

Abstract

This paper presents the design and implementation of NMS, a modular platform for monitoring computer networks and collecting system logs. The system employs a client–server architecture consisting of a multi-threaded C++ server with SQLite database storage, and a Python-based graphical client using PyQt6. The platform uses a simple text-based protocol over TCP for communication, enabling real-time log collection from system sources such as `/var/log/syslog`. Key features include user authentication against the database, automatic log monitoring with background threads, and a modern dark-themed graphical interface designed with Qt Designer. This work demonstrates how traditional network monitoring can be implemented efficiently using modern C++ threading, Python GUI frameworks, and lightweight embedded databases.

Keywords: Network monitoring, Syslog, C++, Python, PyQt6, SQLite, Client–Server, Multi-threading

1 Introduction

The proliferation of connected devices and the increasing complexity of IT infrastructure have made network monitoring an essential component of system administration. Log collection and analysis provide crucial visibility into system health, security events, and operational issues.

The goal of this project is to build a functional network monitoring platform based on a client–server architecture. The system addresses the fundamental requirements of centralised log collection, user authentication, and real-time visualisation of monitoring data.

The platform operates on the following principles. First, the server acts as a central collection point for logs from multiple clients. Second, each client monitors its local system logs and transmits them to the server. Third, user authentication ensures that only authorised users can access the system. Fourth, a graphical interface provides real-time visibility into collected data.

1.1 Inspiration: Wazuh Security Platform

This project draws significant inspiration from **Wazuh** [7], an open-source security monitoring platform widely used in enterprise environments. Wazuh provides comprehensive capabilities for threat detection, integrity monitoring, incident response, and compliance management.

Key aspects adopted from Wazuh's architecture include:

- **Agent-Server Model**: Like Wazuh, NMS uses lightweight agents (clients) that collect local logs and forward them to a central server for processing and storage.
- **C++ Implementation**: Wazuh's core components are written in C++, demonstrating that high-performance security software benefits from low-level system access and efficient memory management.
- **Syslog Integration**: Wazuh's log collector (`logcollector`) monitors system log files such as `/var/log/syslog`, `/var/log/auth.log`, and custom paths—a pattern replicated in NMS.
- **Protocol Design**: The command-based protocol (REGISTER, HEARTBEAT, events) mirrors Wazuh's agent-manager communication patterns.
- **SQLite for Local Storage**: Both systems use embedded SQLite databases for lightweight, file-based storage without external database dependencies.

Being open-source (GPL v2), Wazuh's codebase [8] served as a valuable reference for understanding production-grade security software architecture. The modular separation between log collection, command processing, and database storage in NMS directly reflects patterns observed in Wazuh's source code.

While Wazuh is a complete enterprise solution with advanced features (OSSEC rules, Elastic integration, compliance dashboards), NMS represents a simplified educational implementation demonstrating the core concepts of network monitoring systems.

2 Applied Technologies

The technology stack was carefully selected to satisfy functional requirements (network monitoring, log collection, user interface) and non-functional requirements (performance, simplicity, cross-platform compatibility).

C++17 serves as the implementation language for the server component. C++ provides deterministic memory management, high performance, and direct access to system APIs essential for network programming and multi-threading. The use of modern C++17 features (structured bindings, `std::thread`, `std::mutex`) ensures code clarity without sacrificing performance.

Python 3 with **PyQt6** implements the graphical client. Python's simplicity enables rapid development of the user interface, while PyQt6 (the Python bindings for Qt 6) provides a mature, cross-platform GUI framework with native look and feel.

Qt Designer was used to visually design the user interface. The tool generates `.ui` XML files that are loaded at runtime by PyQt6, enabling separation of UI design from application logic.

TCP Sockets provide the transport layer for client–server communication. TCP guarantees ordered, reliable delivery of monitoring events. The implementation uses standard POSIX socket APIs on Linux.

Text-Based Protocol serves as the application-level message format. Simple newline-delimited text commands (e.g., `LOGIN`, `REGISTER`, `HEARTBEAT`, `BATCH_EVENT`) are human-readable and easily debugged using tools like Wireshark.

SQLite provides embedded database functionality on the server. The database stores user credentials (table `Utilizatori`) and collected logs (table `Loguri`). SQLite requires no separate database server, simplifying deployment.

Thread Pool architecture enables the server to handle multiple simultaneous client connections efficiently. A pool of worker threads processes incoming connections, avoiding the overhead of thread creation per connection.

3 Application Structure

The system comprises two main components: the C++ server and the Python client. Figure 1 illustrates the high-level architecture.

3.1 Server Component

The server runs as a native Linux binary and performs the following functions:

- **Connection Management**: A main listener thread accepts incoming TCP connections on port 8080 and dispatches each connection to a worker thread from the pool.
- **Protocol Handling**: Worker threads parse incoming text commands, validate their structure, and route them to the appropriate processing function based on command type (LOGIN, REGISTER, HEARTBEAT, BATCH_EVENT).

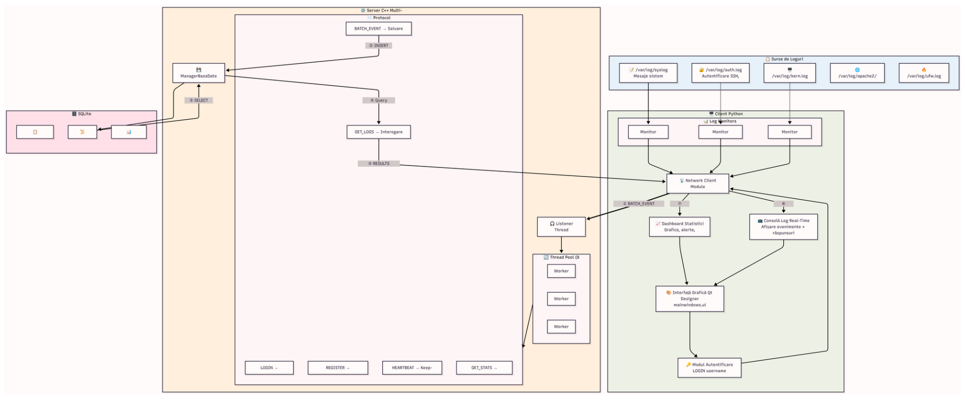

Figure 1: High-level architecture of NMS showing the client–server structure.

- **Authentication**: The LOGIN command verifies username and password against the `Utilizatori` table in SQLite using the `Autentificare()` method.
- **Log Storage**: BATCH_EVENT commands are processed by the `SalveazaLog()` method, which inserts the log message into the `Loguri` table with a timestamp.
- **Response Generation**: Results are serialised as text responses (e.g., `ACK OK Login successful`) and transmitted back to the client.

The core thread pool mechanism is implemented using modern C++17 synchronization primitives:

Listing 1: Thread Pool Worker Implementation (Logic_Server.cpp).

```cpp
void Server::FunctieThreadLucrator() {
  while (true) {
    function<void()> sarcina;
    {
      // Blocheaza mutex-ul inainte de a accesa coada
      unique_lock<mutex> blocare(mutex_coada);

      // Asteapta (doarme) pana cand apare o sarcina in
          coada
      conditie_trezire.wait(blocare, [this]() {
        return !coada_sarcini.empty() || opreste_serverul
            ;
      });

      if (opreste_serverul && coada_sarcini.empty())
          return;

```

```
            // Ia sarcina din fata cozii
            sarcina = move(coada_sarcini.front());
            coada_sarcini.pop();
        }
        // Executa sarcina in afara blocarii
        sarcina();
    }
}
```

The server source files are organised as follows:

- `Main_Server.cpp` – Entry point
- `Logic_Server.cpp/h` – Connection handling and thread pool
- `Commands_Processing.cpp/h` – Protocol command handlers
- `db.cpp/h` – SQLite database wrapper (`ManagerBazaDate` class)

3.2 Client Component

The client is a Python application with a graphical interface built using PyQt6. It provides the following functionality:

- **Connection**: Establishes TCP connection to the server using Python's `socket` module.
- **Authentication**: Sends LOGIN command with username and password entered in the UI. Only enables features upon successful authentication.
- **Log Monitoring**: A background thread (`LogMonitorThread`) continuously reads `/var/log/syslog` and sends new entries to the server using BATCH_EVENT commands.
- **Manual Commands**: Buttons for REGISTER and HEARTBEAT commands.
- **Real-Time Display**: The log console displays sent commands, received responses, and monitored log entries in real time.

The log monitoring logic runs in a separate background thread to keep the UI responsive:

Listing 2: Log Monitoring Thread (client_gui.py).

```python
class LogMonitorThread(QThread):
    log_signal = pyqtSignal(str)

    def run(self):
        # ... check file access ...
        with open(self.log_path, 'r') as f:
            f.seek(0, 2) # Jump to end of file
            while self.running and self.client.connected:
                line = f.readline()
                if line:
                    clean = line.strip()
                    if clean:
                        # Send to server via TCP
                        self.client.send(f"BATCH_EVENT {clean}")
                        self.log_signal.emit(f"[LOG] {clean[:80]}...")
                else:
                    time.sleep(0.5)
```

The client source files include:

- `client_gui.py` – Main application with PyQt6 UI logic
- `mainwindows.ui` – Qt Designer UI definition

Figure 2 shows the internal architecture of the client application.

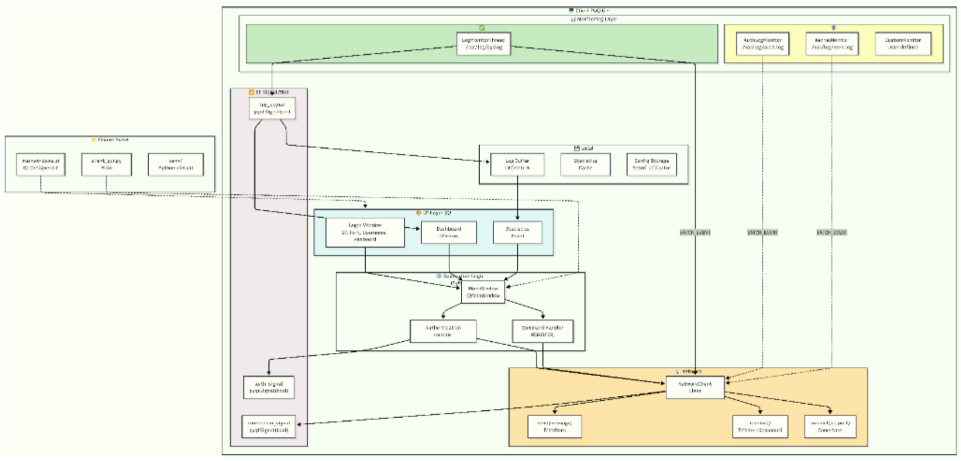

Figure 2: Internal architecture of the Python client showing UI layer, network layer, monitoring threads, and signal/slot system.

3.3 Database Schema

The SQLite database (`nms_romania.db`) contains two tables:

Listing 3: SQL Schema Creation (C++ embedded string).

```sql
CREATE TABLE IF NOT EXISTS Utilizatori (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nume_utilizator TEXT UNIQUE NOT NULL,
    parola TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS Loguri (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    nume_utilizator TEXT,
    mesaj TEXT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Inseram un user default 'admin' / 'admin' daca nu
   exista
INSERT OR IGNORE INTO Utilizatori (nume_utilizator,
   parola)
    VALUES ('admin', 'admin');
```

3.4 Data Flow Summary

The complete data flow from log source to database proceeds as follows:

1. Client connects to server on TCP port 8080.
2. Server sends welcome message.
3. Client sends LOGIN with credentials from UI.
4. Server verifies credentials against SQLite and responds with ACK.
5. Upon successful login, client starts log monitoring thread.
6. Monitor thread reads new lines from `/var/log/syslog`.
7. Each line is sent to server as BATCH_EVENT command.
8. Server stores the log in SQLite via `SalveazaLog()`.
9. Server responds with ACK for each event.
10. Client displays activity in the log console.

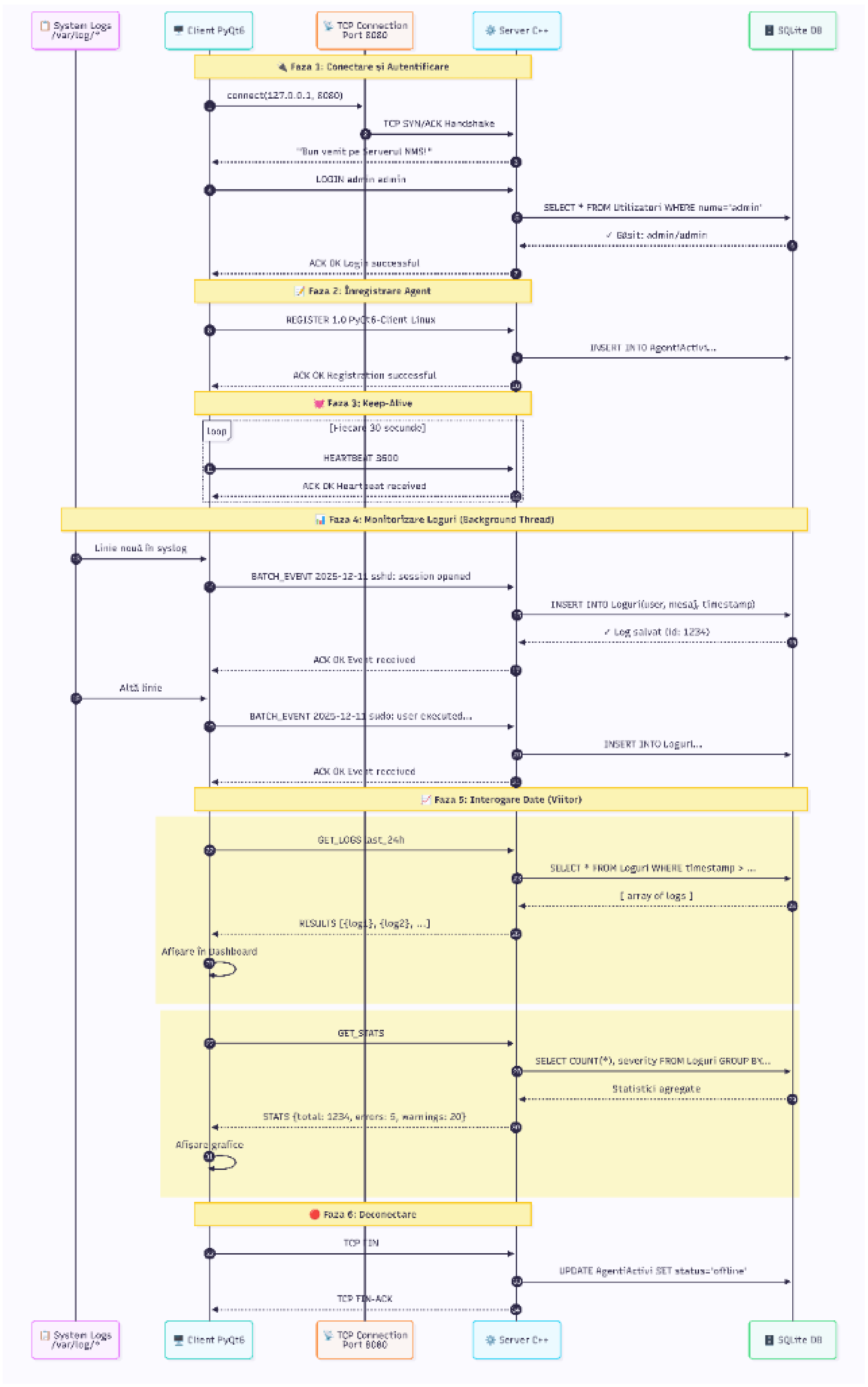
System Logs /var/log/*
Client PyQt6
TCP Connection Port 8080
Server C++
SQLite DB
Faza 1: Conectare și Autentificare
connect(127.0.0.1, 8080)
TCP SYN/ACK Handshake
"Bun venit pe Serverul NMS!"
LOGIN admin admin
SELECT * FROM Utilizatori WHERE nume='admin'
✓ Găsit: admin/admin
ACK OK Login successful
Faza 2: Înregistrare Agent
REGISTER 1.0 PyQt6-Client Linux
INSERT INTO AgentiActivi...
ACK OK Registration successful
Faza 3: Keep-Alive
loop
[Fiecare 30 secunde]
HEARTBEAT 3600
ACK OK Heartbeat received
Faza 4: Monitorizare Loguri (Background Thread)
Linie nouă în syslog
BATCH_EVENT 2025-12-11 sshd: session opened
INSERT INTO Loguri(user, mesaj, timestamp)
✓ Log salvat (id: 1234)
ACK OK Event received
Altă linie
BATCH_EVENT 2025-12-11 sudo: user executed...
INSERT INTO Loguri...
ACK OK Event received
Faza 5: Interogare Date (Viitor)
GET_LOGS last_24h
SELECT * FROM Loguri WHERE timestamp > ...
[array of logs]
RESULTS [{log1}, {log2}, ...]
Afișare în Dashboard
GET_STATS
SELECT COUNT(*), severity FROM Loguri GROUP BY...
Statistici agregate
STATS {total: 1234, errors: 5, warnings: 20}
Afișare grafice
Faza 6: Deconectare
TCP FIN
UPDATE AgentiActivi SET status='offline'
TCP FIN-ACK

Figure 3: Sequence diagram showing the complete data flow: connection, authentication, log monitoring, and database storage.

4 Communication Protocol

Communication uses a simple text-based protocol over TCP. Each message is a single line terminated by a newline character (`\n`).

4.1 Message Format

Messages follow the format: `COMMAND [arguments...]`

Listing 4: Message format examples.

```
LOGIN admin admin
REGISTER 1.0 PyQt6-Client Linux
HEARTBEAT 3600
BATCH_EVENT 2025-12-11T16:00:00 sshd: Failed password for
    root
ACK OK Login successful
ACK FAIL Invalid credentials
```

4.2 Command Types

The protocol defines five command types:

LOGIN: Sent by client to authenticate.

```
LOGIN <username> <password>
```

REGISTER: Sent by client to register as a monitoring agent.

```
REGISTER <version> <hostname> <os>
```

HEARTBEAT: Sent periodically to indicate liveness.

```
HEARTBEAT <uptime_seconds>
```

BATCH_EVENT: Sends a log entry to the server.

```
BATCH_EVENT <log_message>
```

ACK: Server response acknowledging a command.

```
ACK <status> <message>
```

4.3 Port Assignment

Table 1: Network port assignment.

Port	**Protocol**	**Direction**	**Purpose**
8080	TCP	Client → Server	Primary communication

4.4 Wireshark Analysis

The text protocol enables easy debugging with Wireshark. Using the filter `tcp.port == 8080` on the loopback interface (`lo`), the complete conversation can be observed:

Listing 5: Example Wireshark TCP stream.

```
Bine ai venit la server!
LOGIN admin admin
ACK OK Login successful
REGISTER 1.0 PyQt6-Client Linux
ACK OK Registration successful
BATCH_EVENT 2025-12-11T16:00:00 sshd: session opened
ACK OK Event received
```

4.5 Connection State Machine

Figure 4 shows the state transitions for a client connection lifecycle, from initial connection through authentication, monitoring, and disconnection.

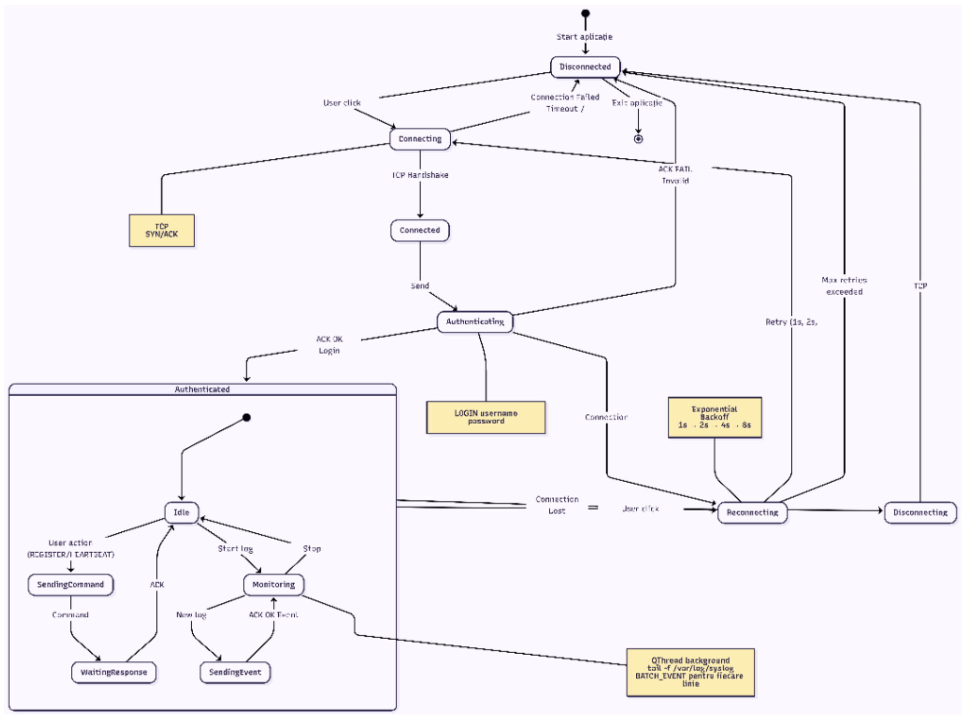

Figure 4: State machine diagram showing client connection lifecycle: Disconnected, Connecting, Authenticating, Authenticated (with Monitoring substates), Reconnecting, and Disconnecting.

5 Real World Use Cases

This section describes six operational scenarios—three successful and three involving system failures—demonstrating the platform's utility in a production environment.

5.1 Successful Execution Scenarios

1. Centralized Security Auditing (Incident Detection)
Use Case: A security administrator needs to investigate a potential breach on the "Finance-Server" node.
Execution: The administrator launches the NMS client and authenticates successfully. They observe the real-time stream of log messages. When the remote node generates `sshd` failure logs, these are immediately propagated to the central dashboard.
Result: The administrator identifies the source IP of the attacker in real-time, allowing for immediate remediation. The system successfully aggregates distributed intelligence into a single view.

2. Infrastructure Scalability (New Node Provisioning)
Use Case: A DevOps engineer adds a new web server to the existing fleet and requires immediate monitoring status.
Execution: The engineer deploys the NMS Agent on the new server. Upon startup, the agent automatically initiates the registration process, transmitting its OS details (Linux Ubuntu) and hostname.
Result: The server accepts the registration without requiring a restart or manual configuration update. The new node immediately begins contributing heartbeat data, verifying the successful expansion of the monitoring perimeter.

3. Continuous Availability Monitoring (Heartbeat Tracking)
Use Case: Network operations staff need to verify that a critical gateway remains online during a weekend maintenance window.
Execution: The agent is configured to send periodic heartbeat signals. The server processes these keep-alive messages continuously.
Result: The server records the periodic signals in the database. The operations team can visually confirm the "Alive" status of the gateway throughout the maintenance window, ensuring Service Level Agreements (SLAs) are met.

5.2 Failure Scenarios and Handling

1. Unauthorized Access Attempt (Insider Threat)
Use Case: An unprivileged user attempts to gain access to the NMS dashboard to delete log evidence.
Execution: The user launches the client and attempts to guess the administrator password, sending invalid credentials.

System Response: The server performs a check against the `Utilizatori` table and returns a failure response. The client application enforces a strict lockout, keeping the "Monitoring" and "History" features disabled. The security of the stored logs is preserved despite the attempted breach.

2. Central Server Outage (Network Partition)
Use Case: A remote branch office loses VPN connectivity to the main datacenter where the NMS Server resides.
Execution: The branch agent attempts to transmit a batch of syslog entries, but the TCP handshake fails due to the network path being down.
System Response: Instead of crashing, the client catches the network exception. It logs a local error state and alerts the user via the UI. The agent enters a "Disconnected" state, preventing data loss attempts until connectivity is restored.

3. Malformed Data Injection (Integrity Protection)
Use Case: A compromised host attempts to crash the monitoring server by sending corrupted or non-compliant protocol messages.
Execution: The malicious agent sends a string that violates the protocol format (e.g., missing command headers or binary garbage).
System Response: The server's command parser detects the anomaly. It classifies the message as unknown or handles the parsing exception safely. The server drops the invalid packet, logs the error to standard output, and maintains the stability of the main thread pool for other legitimate clients.

6 Conclusion

This work presents NMS, a functional client–server platform for network monitoring and log collection. The system successfully demonstrates centralised log collection, user authentication, and real-time visualisation.

The architecture provides clear separation of concerns: the C++ server handles network connections and database operations with high performance, while the Python client provides a user-friendly graphical interface. The text-based protocol simplifies debugging and integration.

6.1 Potential Improvements

Several enhancements could extend the platform's capabilities:

- **Password Hashing**: Store hashed passwords instead of plaintext.
- **TLS Encryption**: Add SSL/TLS for secure communication over untrusted networks.
- **Log Filtering**: Allow clients to filter logs by severity, source, or keyword.

- **Dashboard Statistics**: Display charts showing log volume, error rates, and trends.
- **Multiple Log Sources**: Support monitoring of additional log files beyond syslog.
- **User Management**: Add ability to create and manage multiple user accounts.

References

[1] Stevens, W.R.: TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley (1994).

[2] SQLite Consortium: SQLite Documentation. `https://www.sqlite.org/docs.html`. Accessed: 11 December 2025.

[3] Riverbank Computing: PyQt6 Documentation. `https://www.riverbankcomputing.com/static/Docs/PyQt6/`. Accessed: 11 December 2025.

[4] The Qt Company: Qt 6 Documentation. `https://doc.qt.io/qt-6/`. Accessed: 11 December 2025.

[5] IEEE: POSIX.1-2017 – Portable Operating System Interface. IEEE Std 1003.1 (2017).

[6] ISO: ISO/IEC 14882:2017 – Programming Language C++. International Organization for Standardization (2017).

[7] Wazuh, Inc.: Wazuh – The Open Source Security Platform. `https://wazuh.com`. Accessed: 11 December 2025.

[8] Wazuh, Inc.: Wazuh GitHub Repository. `https://github.com/wazuh/wazuh`. Accessed: 11 December 2025.

[9] Gerhards, R.: The Syslog Protocol. RFC 5424, IETF (2009).

[10] Olusola, O. et al.: Design and Implementation of Network Monitoring System for Campus Infrastructure Using Software Agents. In: Journal of Computer Networks and Communications (2021).