Technical Report: NetworkDevicesMonitor

Isari Andrew

No Institute Given

1 Introduction

The primary objective of this project is to develop a high-performance, secure Network Operations (NetOps) platform designed to automate the discovery and monitoring of network infrastructure. The vision relies on a centralized Client-Server architecture where distributed "active agents" perform autonomous network scanning and report telemetry to a central controller. This solution aims to replace fragmented manual inventory processes with a real-time, persistent database system, ensuring data consistency and operational security through encrypted communication channels.

2 Applied Technologies

To achieve high concurrency and security without reliance on heavy frameworks, the following technologies were selected:

– **C++17 & CMake:** Chosen for deterministic memory management and strict control over build dependencies. The project enforces a separation of concerns by linking `Qt6::Widgets` strictly to the Client and `SQLite3` strictly to the Server.
– **Raw TCP Sockets & Epoll:** The Server utilizes the Linux `epoll` mechanism (Reactor Pattern) with non-blocking sockets (`O_NONBLOCK`). This allows a single thread to manage thousands of concurrent connections efficiently.
– **UDP (User Datagram Protocol):** Utilized by the Client's **Scanner Thread** for high-speed network probing and service discovery. UDP allows the agent to fire "fire-and-forget" packets to scan subnets rapidly without waiting for TCP connection overhead.
– **OpenSSL (TLS 1.3):** Security is implemented by manually integrating the OpenSSL C API with raw file descriptors. This ensures end-to-end encryption for all control traffic while maintaining low-level control over the handshake state machine.
– **SQLite:** Selected for server-side persistence due to its reliability and zero-configuration deployment, utilizing prepared statements (`sqlite3_prepare_v2`) to prevent injection attacks.
– **Qt6 Widgets:** Provides the client-side graphical interface. It is decoupled from the networking logic via a `QTimer` bridging mechanism, ensuring the UI remains responsive during network operations.

3 Application Structure

The system implements a **Secure, Asynchronous Agent-Controller Architecture**.

3.1 Core Modeling Concepts

1. **The Reactor Pattern (Server):** The server does not block on I/O. It waits for events (read/write readiness) and delegates high-latency tasks (Database I/O) to a **Worker Thread Pool** via a thread-safe queue. This ensures the main loop remains responsive to new handshakes.
2. **The Active Agent (Client):** The Client is multi-threaded. A background **Scanner Thread** actively reads system tables (e.g., `/proc/net/arp`) and utilizes **UDP/Raw Sockets** to probe the network for active devices. A separate **Network Thread** manages the persistent, encrypted TCP session with the Server.
3. **Buffer Management:** A custom `ByteBuffer` class handles TCP stream fragmentation, accumulating partial bytes until a complete protocol header and payload are available.

3.2 System Architecture Diagram

The following diagram illustrates the logical flow of data from the Client's UI, through the encrypted network layer, to the Server's event loop, and finally to the persistent storage.

4 Communication Protocol

The platform utilizes a custom **Binary Application Layer Protocol** designed for bandwidth efficiency and strict byte alignment.

4.1 Header Specification

Every transmission begins with a fixed-size header structure (16 bytes), serialized in **Network Byte Order (Big-Endian)**:

- **Magic Number (2 bytes):** `0xA5A5` (Sanity check).
- **Version (2 bytes):** `0x0001` (Protocol versioning).
- **Packet Type (2 bytes):** Identifies the operation (e.g., `0x0010` for LOGIN, `0x0020` for REPORT_DATA).
- **Payload Length (4 bytes):** The exact size of the subsequent data block.
- **Reserved (6 bytes):** Padding for future expansion.

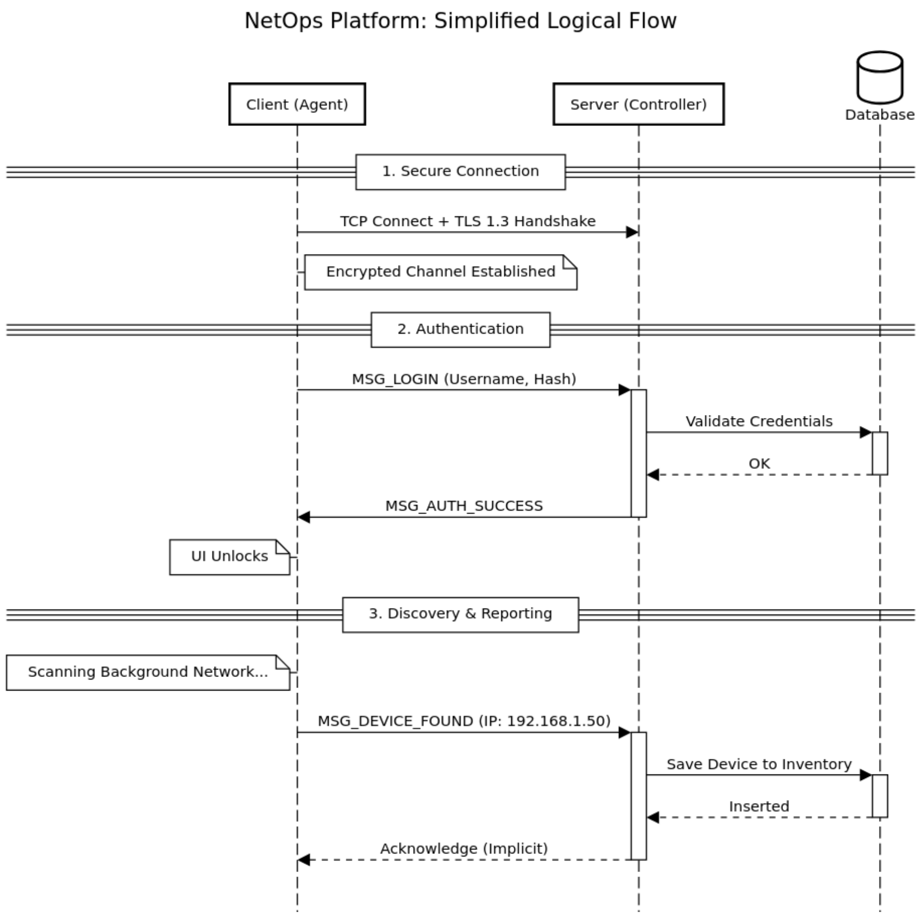

Fig. 1. Logical Flow: Client UI, Network Thread, Epoll Loop, and Worker Thread

5 Use Cases

5.1 Successful Execution Scenarios

1. **Secure Agent Login:** A Client initiates a connection. The Server accepts the TCP handshake and triggers the TLS 1.3 handshake. Once the channel is secure, the Client sends a `LOGIN` packet containing a hashed credential. The Server's Worker Thread validates the hash against the `users` table and returns `AUTH_SUCCESS`.
2. **Device Discovery Report:** The Client's Scanner Thread identifies a new host via a UDP probe. It queues a `DEVICE_FOUND` message. The Network Thread picks it up, encrypts it, and pushes it to the Server. The Server parses the payload and performs an `INSERT OR IGNORE` into the `devices` inventory table.

3. **Heartbeat Synchronization:** To maintain the NAT mapping, the Client sends a `PING` packet every 60 seconds. The Server updates the `last_seen` timestamp for that agent in the database and responds with `PONG`.

5.2 Failure Scenarios

1. **Fragmentation/Partial Read:** The Server receives only the first 10 bytes of a 16-byte header. The `ByteBuffer` logic detects `buffer.size() < sizeof(Header)`, retains the data, and returns control to `epoll_wait`. Processing resumes only when the remaining bytes arrive.
2. **TLS Handshake Failure:** A client attempts to connect without the correct SSL Certificate. OpenSSL returns an error during `SSL_accept`. The Server detects the alert, logs "Handshake Failed," and closes the file descriptor immediately to free resources.
3. **Database Busy/Lock:** Two agents report data simultaneously. The Server's Worker Thread attempts to write to SQLite but encounters a lock. The thread captures the `SQLITE_BUSY` error, waits for a backoff period, and retries the transaction without crashing the application.

6 Conclusions

The proposed solution successfully implements a secure, scalable NetOps platform. By avoiding high-level networking wrappers (like QtNetwork) in favor of raw `epoll`, UDP, and OpenSSL, the system achieves fine-grained control over concurrency and memory usage.

Potential Improvements:

- **Database Migration:** Transitioning from SQLite to PostgreSQL to support multi-writer clustering for larger networks.
- **Packet Capture Analysis:** Extending the Scanner Thread to perform basic PCAP analysis for traffic anomaly detection.

References

1. Stevens, W. R., Rago, S. A.: Advanced Programming in the UNIX Environment, 3rd ed. Addison-Wesley (2013)
2. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (2018)
3. Springer LNCS Guidelines, `https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines`