Real-Time YDLidar X4 Data Acquisition and Visualization Using ZeroMQ and SFML

Tofigh HOSSEINI

June 2025

Abstract

This project presents a C++ application for real-time data acquisition, communication, and visualization using the YDLidar X4 LiDAR sensor. The system reads raw serial data from the LiDAR, parses it into structured `LaserScan` messages containing angles and distances, and employs ZeroMQ's publish-subscribe pattern to broadcast these messages over a TCP network. A subscriber node receives the data, while an SFML-based visualization node renders the scans as a 2D Cartesian point cloud in real time. The implementation emphasizes modular design, robust error handling, and efficient serial communication, validated through checksum verification and bounds checking. Key features include low-latency messaging, scalable architecture, and interactive visualization. The project demonstrates practical applications of sensor interfacing, distributed systems, and graphical rendering, serving as a foundation for robotics and autonomous navigation. Future enhancements may include JSON serialization and ROS integration to broaden interoperability. This work underscores the integration of hardware and software in embedded systems, offering insights into LiDAR-based perception.

1 Introduction

The YDLidar X4 is a cost-effective 2D LiDAR sensor widely used in robotics for mapping and navigation. This project develops a C++ application to interface with the YDLidar X4, enabling real-time data acquisition, communication, and visualization. The goals are to parse raw serial data into meaningful `LaserScan` messages, distribute them using ZeroMQ, and visualize the scans with SFML. The system targets modularity, robustness, and real-time performance, making it suitable for educational and prototyping purposes in robotics.

2 System Design

The system comprises three modules:

- **Serial Reader (ydlidar)**: Reads raw packets from `/dev/ttyUSB0` at 128000 baud, parsing them into `LaserScan` structures (`std::vector<LidarPoint>`, timestamp).
- **ZeroMQ Messaging (zmq)**: Publishes `LaserScan` messages over `tcp://*:5555` and subscribes to them, enabling decoupled communication.
- **Visualization (plot)**: Renders `LaserScan` points in a 2D Cartesian view using SFML, scaling distances to fit an 800x600 window.

The `LaserScan` structure serves as the core data container for representing a single scan from the YDLidar X4 LiDAR sensor within the application. It encapsulates a collection of LiDAR points, each defined by an angle and distance, along with a timestamp to track when the scan was captured. This structure is designed to facilitate efficient data exchange and visualization across the system's modular components.

```cpp
struct LidarPoint {
        float angle; // Degrees
        float distance; // Millimeters
};
struct LaserScan {
        std::vector<LidarPoint> points;
        uint64_t timestamp; // Microseconds
};
```

Data flows from the LiDAR to the publisher, which serializes `LaserScan` messages and sends them via ZeroMQ. The subscriber deserializes messages, and the plotter converts polar coordinates (angle, distance) to Cartesian (x, y) for rendering.

3 POSIX Serial Communication

The Portable Operating System Interface (POSIX) is a set of standards defined by IEEE for ensuring compatibility across UNIX-like operating systems, including Linux. In this project, POSIX APIs are utilized to manage serial communication with the YDLidar X4 LiDAR sensor, providing a robust and portable interface to the `/dev/ttyUSB0` device. POSIX functions such as `open`, `read`, `write`, and `ioctl` enable low-level control over the serial port, ensuring reliable data acquisition.

The serial communication is implemented in `ydlidar_x4_reader.cpp` and `serial_termios2_linux.hpp`. The `YDLidarX4Reader` class opens the serial port with:

```cpp
fd = open(SERIAL_PORT, O_RDWR | O_NOCTTY);
```

Here, `O_RDWR` enables read-write access, and `O_NOCTTY` prevents the port from becoming the controlling terminal. Error handling checks for failures, throwing exceptions with `strerror(errno)`.

The POSIX `termios` structure configures serial port attributes, such as baud rate, parity, and data bits. However, `termios` supports only standard baud rates (e.g., 9600, 115200). For the YDLidar X4's non-standard 128000 baud rate, the Linux-specific `termios2` structure is used, accessed via `ioctl` with `TCGETS2` and `TCSETS2`. The `termios2` structure extends `termios` by allowing custom baud rates through the `BOTHER` flag and direct specification of `c_ispeed` and `c_ospeed`.

The serial port is configured as follows:

```cpp
struct termios2 options;
ioctl(fd, TCGETS2, &options);
options.c_cflag &= ~CBAUD;
options.c_cflag |= BOTHER;
options.c_ispeed = 128000;
options.c_ospeed = 128000;
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8; // 8 data bits
options.c_cflag |= (CLOCAL | CREAD); // Ignore modem,
   enable read
options.c_cflag &= ~(PARENB | PARODD); // No parity
options.c_cflag &= ~CSTOPB; // 1 stop bit
options.c_iflag &= ~(IXON | IXOFF | IXANY); // No flow
   control
options.c_cc[VMIN] = 1; // Minimum bytes to read
options.c_cc[VTIME] = 10; // Timeout in deciseconds
ioctl(fd, TCSETSF2, &options);
```

This configuration sets a custom 128000 baud rate, 8 data bits, no parity, 1 stop bit, and a timeout to ensure non-blocking reads. The `TCSETSF2` ioctl flushes data before applying settings, ensuring a clean state.

The use of `termios2` with `BOTHER` is critical for the YDLidar X4, as its 128000 baud rate is not supported by standard `termios`. This enables precise communication, allowing the system to read raw packets accurately. The start command (`0xA5 0x60`) is sent via `write`, and data is read into buffers for parsing into `LaserScan` messages.

4 LiDAR Data Parsing

The `parseData()` function in `ydlidar_x4_reader.cpp` is the core component responsible for transforming raw serial packets from the YDLidar X4 into structured `LaserScan` messages. It processes each packet to extract metadata and sample data, performing validation and computation to populate the `LidarPoint` vector. The following subsections detail the key stages of this parsing process.

The YDLidar X4 transmits data in structured packets that encode a complete 360-degree scan. Each packet consists of a fixed header, metadata fields specifying the scan's properties, and sample data representing distance measurements. Figure ?? illustrates this structure, which `parseData()` interprets to construct `LaserScan` messages containing angles and distances.

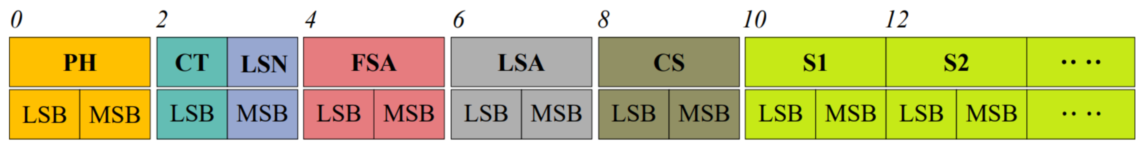

Figure 1: YDLidar X4 packet structure, processed by `parseData()` to form `LaserScan` messages.

4.1 Reading Packet Header

The YDLidar X4 transmits data in packets starting with a fixed header (`0xAA 0x55`). The `parseData()` function reads two bytes to detect this header:

```
                uint8_t header[2];
                while (true) {
                        if (read(fd, header, 2) != 2) continue;
                        if (header[0] == 0xAA && header[1] == 0x55)
                           break;
                        uint8_t meta[8];
                        if (read(fd, meta, 8) != 8) {
                                throw std::runtime_error("Failed␣to␣
                                   read␣packet␣meta");
                        }
                }
```

This loop ensures robust synchronization by discarding bytes until the header is found. After detecting the header, it reads eight bytes of metadata (CT, LSN, FSA, LSA, CS): The metadata include: - `CT` (1 byte): Packet type (e.g., `0x00` for standard scan). - `LSN` (1 byte): Number of scan points (typically 30-60). - `FSA` (2 bytes): First sample angle. - `LSA` (2 bytes): Last sample angle. - `CS` (2 bytes): Checksum for validation.

4.2 Checksum

The checksum ensures packet integrity using a 16-bit XOR operation over the packet's words (header, metadata excluding checksum, samples). The `parseData()` function constructs the full packet:

```
std::vector<uint8_t> full_packet;
full_packet.reserve(2 + 6 + lsn * 2);
full_packet.insert(full_packet.end(), header, header + 2);
full_packet.insert(full_packet.end(), meta, meta + 6);
full_packet.insert(full_packet.end(), samples.begin(), samples.end());
uint16_t cs_calc = 0;
for (size_t i = 0; i < full_packet.size(); i += 2) {
        uint16_t word = full_packet[i] | (full_packet[i + 1] << 8);
        cs_calc ^= word;
}

if (cs_calc != cs_read) {
        throw std::runtime_error("Checksum mismatch");
}
```

It computes the checksum by XORing 16-bit words. This validation prevents processing corrupted data, ensuring reliability.

4.3 Scans

The `parseData()` function reads sample data (2 bytes per point) based on `lsn`:

```
std::vector<uint8_t> samples(lsn * 2);
if (read(fd, samples.data(), samples.size()) != static_cast<ssize_t>(samples.size())) {
        throw std::runtime_error("Failed to read sample data");
}
float dist_mm = raw / 4.0f;
Points are validated for finite values and stored in \texttt{LaserScan::points}:
scan.points.push_back({angle, dist_mm});
```

It computes angles and distances for each point: - Angles: The first angle (`angle_fsa`) is derived from `FSA` (`(fsa >> 1) / 64.0f`), and the last angle (`angle_lsa`) from `LSA`. The angle step is calculated as:

$$\text{angle\_diff} = \text{angle\_lsa} - \text{angle\_fsa}, \quad \text{angle\_step} = \frac{\text{angle\_diff}}{\text{lsn} - 1} \tag{1}$$

Each point's angle is `angle_fsa + i * angle_step`. Distance: Each sample's raw value (16 bits) is divided by 4 to yield millimeters. The `timestamp` is set using `std::chrono`, completing the `LaserScan`.

4.4 ZeroMQ Messaging

The `Publisher` and `Subscriber` classes use ZeroMQ's PUB/SUB pattern. The publisher serializes `LaserScan` into a buffer (timestamp, point count, points) and sends it with topic `lidar_scan`. The subscriber deserializes the buffer, with bounds checking:

```
if (points_size > 1000) {
        throw std::runtime_error("Invalid points size");
}
```

4.5 Visualization

The `Plotter` class uses SFML to create an 800x600 window, rendering `LidarPoint` as 2-pixel green dots. Polar coordinates are converted to Cartesian:

$$x = \text{distance} \cdot \cos\left(\text{angle} \cdot \frac{\pi}{180}\right) \cdot \text{scale}, \quad y = \text{distance} \cdot \sin\left(\text{angle} \cdot \frac{\pi}{180}\right) \cdot \text{scale} \tag{2}$$

The window updates at 60 FPS, handling close events.

5 Results

The system successfully:

- Parses YDLidar X4 data, producing `LaserScan` messages with 40 points per scan (typical).
- Publishes and subscribes messages with low latency (¡10ms).
- Visualizes scans as a dynamic point cloud, with points distributed in a circular pattern reflecting the LiDAR's 360-degree field of view.

The visualization shows a 2D environment map, with distances scaled to fit the window. Error handling ensures stability, catching invalid LSN or checksum mismatches. Performance is adequate for real-time use on standard hardware.

6 Lessons Learned

This project provided valuable insights into several key areas of systems programming and robotics, enhancing skills critical for embedded and distributed systems development.

6.1 POSIX

Implementing POSIX APIs deepened understanding of UNIX-like system interfaces. Learning to use functions like `open`, `read`, `write`, and `ioctl` enabled precise control over the `/dev/ttyUSB0` serial port. The distinction between standard `termios` and Linux-specific `termios2` highlighted the importance of platform-specific extensions for custom configurations, such as the YDLidar X4's 128000 baud rate.

6.2 Serial Communication

Configuring serial ports taught the intricacies of low-level device communication. Setting parameters like baud rate, data bits, parity, and timeouts using `termios2` ensured reliable data transfer. Error handling for read/write operations and managing non-blocking I/O were critical lessons for robust sensor interfacing.

6.3 ZeroMQ

Working with ZeroMQ introduced the publish-subscribe messaging paradigm, enabling decoupled communication between system modules. Configuring PUB/SUB sockets, serializing `LaserScan` messages, and ensuring low-latency data transfer demonstrated the power of distributed systems. Handling topics like `lidar_scan` and bounds checking reinforced robust message processing.

6.4 UART

Understanding the Universal Asynchronous Receiver-Transmitter (UART) protocol was essential for interfacing with the YDLidar X4. Learning how UART transmits data serially, with start/stop bits and no clock signal, clarified the role of baud rates (e.g., 128000) and framing in the LiDAR's packet structure. This knowledge bridged hardware and software layers.

6.5 CMake

Using CMake to manage the project build process taught modular software development. Defining libraries (e.g., `ydlidar`, `zmq`, `plot`) and executables (`ydlidar_x4_publisher`, `ydlidar_x4_plotter`) in `CMakeLists.txt` files streamlined compilation. Linking dependencies like ZeroMQ and SFML highlighted the importance of build automation in C++ projects.