Zenoh Node Implementation

Bridging Modern Pub/Sub Communication with Embedded CAN Systems

Autonomous Vehicle Project Documentation

January 2026

Abstract

This document describes the implementation of a Zenoh communication node within the autonomous vehicle project's CAN subsystem. The vehicle's primary communication architecture is built upon Zenoh, providing flexible pub/sub messaging between high-level systems. This document details the extension of this architecture to include CAN bus support, enabling communication with legacy automotive components and low-level embedded controllers. We cover the integration challenges, including vendoring the zenoh-pico library and implementing custom serialization protocols, while providing an overview of both Zenoh and CAN 2.0 protocols and their fundamental differences.

Contents

1 Introduction

The autonomous vehicle project is built on a Zenoh-based communication architecture, which serves as the backbone for all inter-component messaging. Zenoh was chosen as the primary protocol due to its flexibility, support for various transport mechanisms, and seamless integration with ROS2 ecosystems.

However, many automotive-grade sensors, actuators, and embedded controllers communicate exclusively via CAN bus—the de facto standard in vehicle electronics. To leverage these components without abandoning our Zenoh-centric architecture, we implemented a bridge node that extends Zenoh's reach into the CAN domain.

This document describes the Zenoh node implementation located in `src/CAN/components/zenoh-node`, which serves as a bidirectional bridge between our existing Zenoh network and newly integrated CAN bus peripherals.

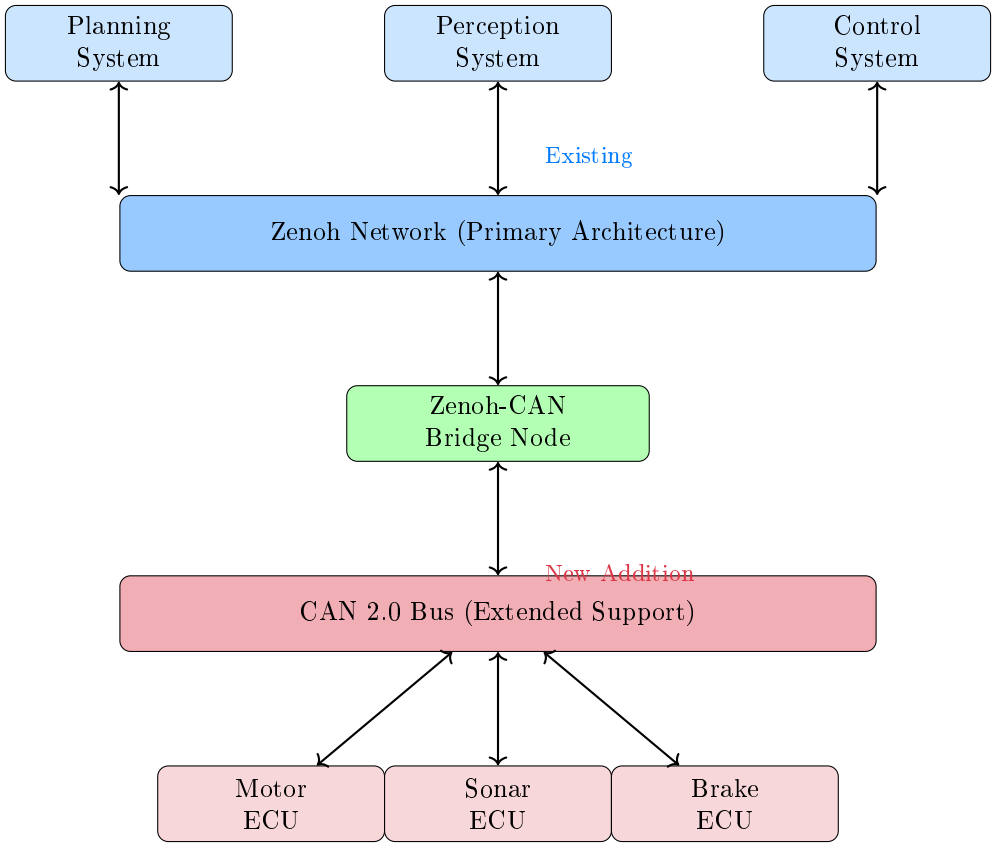

Figure 1: System architecture: Zenoh serves as the primary communication layer, with CAN support added via a bridge node

1.1 Architectural Philosophy

The vehicle's communication architecture follows a **Zenoh-first** approach:

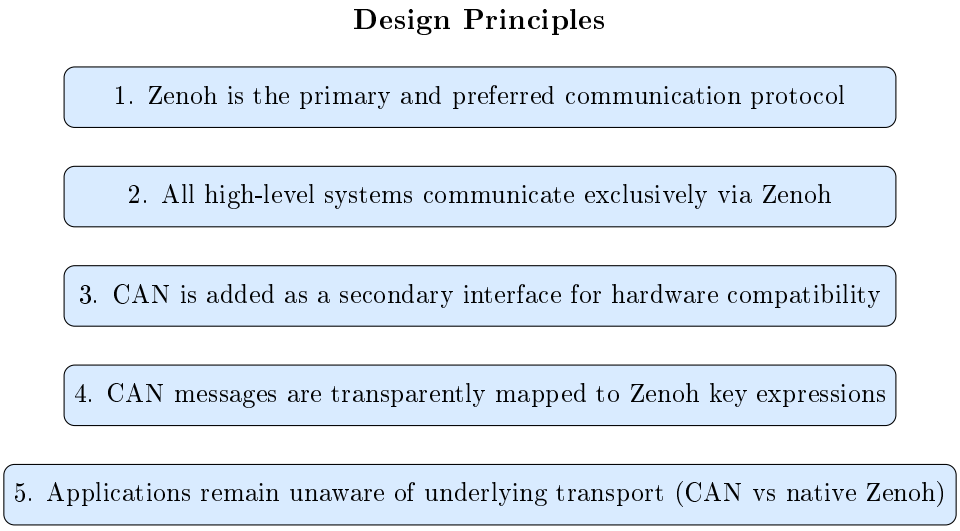

Figure 2: Zenoh-first architectural principles

1.2 Why Add CAN Support?

While Zenoh provides excellent capabilities for modern distributed systems, CAN bus support was added for the following reasons:

- **Hardware Compatibility**: Many automotive-grade sensors and actuators only support CAN
- **Industry Standards**: CAN is mandated for certain automotive safety systems
- **Cost Effectiveness**: CAN-based components are often more affordable and readily available
- **Real-time Guarantees**: CAN provides deterministic timing for safety-critical operations
- **Existing Ecosystem**: Leverage existing CAN-based ECUs without firmware modifications

2 Understanding CAN 2.0

2.1 Overview

Controller Area Network (CAN) is a robust serial communication protocol originally developed by Bosch in the 1980s for automotive applications. CAN 2.0 defines two frame formats:

- **CAN 2.0A**: Standard frame with 11-bit identifier
- **CAN 2.0B**: Extended frame with 29-bit identifier

2.2 CAN Frame Structure

CAN 2.0A Standard Frame:

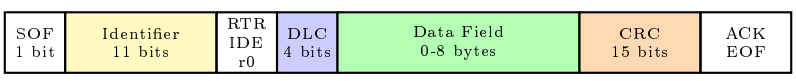

Figure 3: CAN 2.0A standard frame format

2.3 Key CAN Characteristics

Characteristic	Value/Description
Maximum Data Size	8 bytes per frame
Bus Speed	Up to 1 Mbit/s
Arbitration	Priority-based (lower ID = higher priority)
Error Detection	CRC, bit stuffing, frame check
Topology	Linear bus with termination resistors
Maximum Nodes	Typically 32-127 nodes
Maximum Length	∼40m at 1 Mbit/s, ∼1km at 50 kbit/s

Table 1: CAN 2.0 key characteristics

2.4 CAN Bus Topology

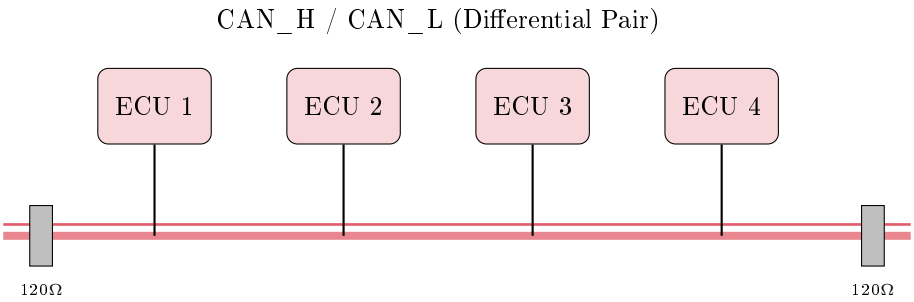

Figure 4: CAN bus topology with termination resistors

3 Understanding Zenoh

3.1 Overview

Zenoh (pronounced "zee-noh") is a modern pub/sub/query protocol designed for edge computing, IoT, and robotics applications. Developed by ZettaScale Technology, Zenoh provides:

- **Unified communication**: Pub/Sub, Query/Reply, and distributed storage
- **Location transparency**: Data can be accessed regardless of where it resides
- **Protocol flexibility**: Works over TCP, UDP, WebSocket, serial, and more
- **Minimal overhead**: Designed for constrained devices (zenoh-pico)

3.2 Zenoh Key Concepts

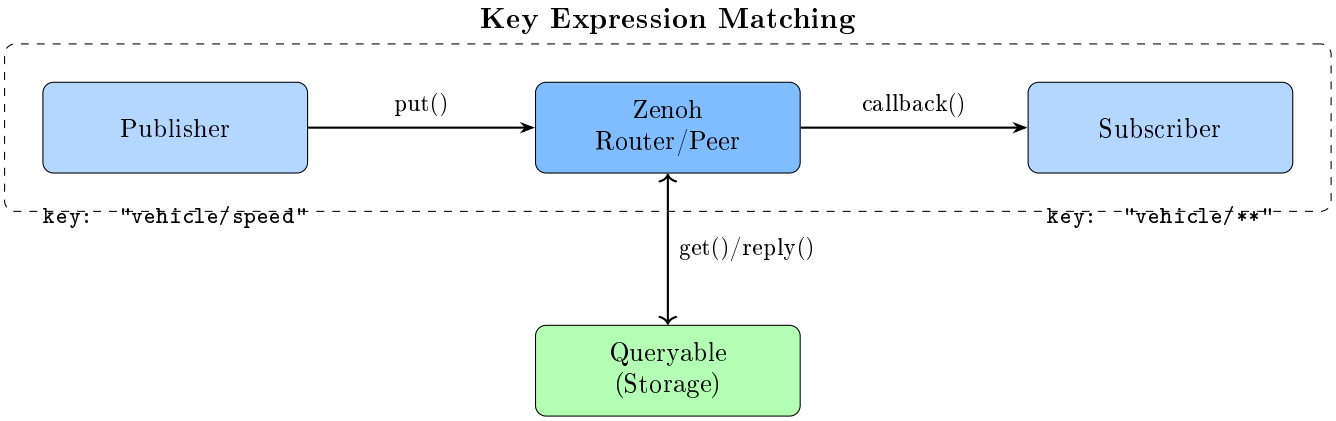

Figure 5: Zenoh communication paradigm

3.3 Key Expressions

Zenoh uses hierarchical key expressions similar to MQTT topics but with more powerful wildcards:

```
# Exact match
vehicle/speed

# Single-level wildcard (*)
vehicle/*/temperature

# Multi-level wildcard (**)
vehicle/**              # Matches all under vehicle/

# Complex expressions
vehicle/sensor/[0-9]+   # Regex support
```

Listing 1: Zenoh key expression examples

4 CAN vs Zenoh: A Comparison

Aspect	CAN 2.0	Zenoh
Data Model	Message-based (ID + Data)	Key-Value (Pub/Sub)
Max Payload	8 bytes	Unlimited (chunked)
Addressing	11/29-bit numeric ID	Hierarchical strings
Discovery	Implicit (ID filtering)	Automatic peer discovery
Topology	Bus (physical)	Mesh/P2P (logical)
QoS	Priority via ID	Configurable reliability
Latency	Microseconds	Milliseconds (typical)

Figure 6: Comparison between CAN 2.0 and Zenoh protocols

4.1 Why Bridge CAN and Zenoh?

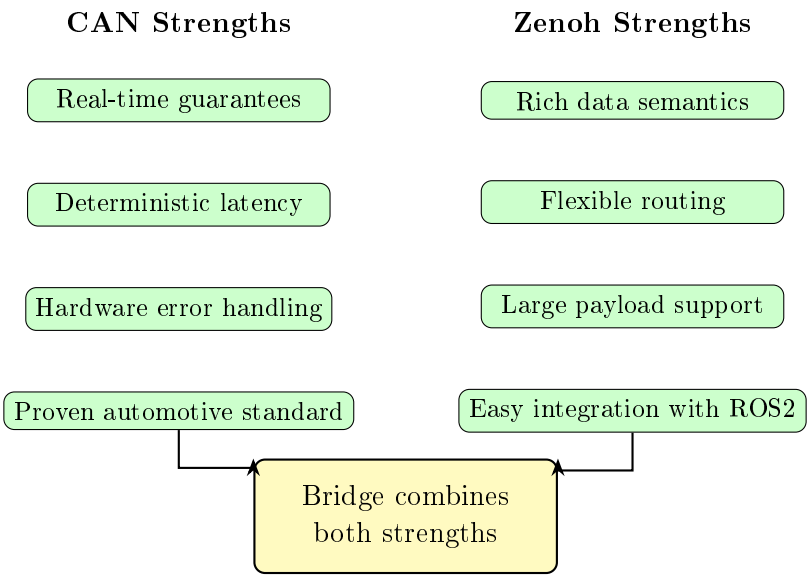

Figure 7: Rationale for bridging CAN and Zenoh

5 Implementation Details

5.1 Project Structure

The Zenoh node is implemented as an ESP-IDF component with the following structure:

```
src/CAN/components/zenoh-node/
|-- CMakeLists.txt          # Build configuration
|-- Kconfig                 # ESP-IDF configuration options
|-- include/
|   |-- zenoh_node.h        # Public API
```

```
|   +-- zenoh_serialize.h   # Serialization interface
|-- src/
|   |-- zenoh_node.c        # Main implementation
|   +-- zenoh_serialize.c   # Custom serialization
+-- vendor/
    +-- zenoh-pico/         # Vendored zenoh-pico library
```

Listing 2: Directory structure of zenoh-node component

5.2 Vendoring zenoh-pico

5.2.1 Why Vendor?

The zenoh-pico library needed to be vendored (included directly in the project) for several reasons:

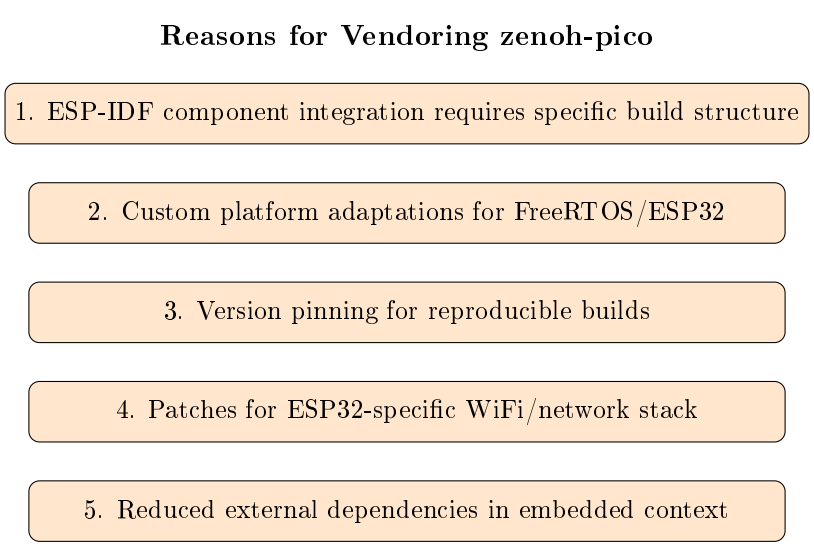

Figure 8: Reasons for vendoring zenoh-pico

5.2.2 Vendoring Process

```
# Clone zenoh-pico at specific version
git clone --depth 1 --branch 0.11.0 \
    https://github.com/eclipse-zenoh/zenoh-pico.git \
    vendor/zenoh-pico

# Remove git history to reduce size
rm -rf vendor/zenoh-pico/.git

# Apply ESP-IDF specific patches
patch -p1 < patches/zenoh-pico-esp-idf.patch
```

Listing 3: Steps to vendor zenoh-pico

5.2.3 CMake Integration

```
idf_component_register(
    SRCS
        "src/zenoh_node.c"
        "src/zenoh_serialize.c"
    INCLUDE_DIRS
        "include"
```

```
        "vendor/zenoh-pico/include"
    REQUIRES
        esp_wifi
        esp_netif
        nvs_flash
    PRIV_REQUIRES
        can_driver
)

# Add zenoh-pico sources
file(GLOB_RECURSE ZENOH_PICO_SRCS
    "vendor/zenoh-pico/src/*.c")
target_sources(${COMPONENT_LIB} PRIVATE ${ZENOH_PICO_SRCS})

# Zenoh-pico configuration
target_compile_definitions(${COMPONENT_LIB} PUBLIC
    ZENOH_ESP32
    Z_FEATURE_PUBLICATION=1
    Z_FEATURE_SUBSCRIPTION=1
    Z_FEATURE_QUERY=1
)
```

Listing 4: CMakeLists.txt for zenoh-node component

5.3 Custom Zenoh Serialization

5.3.1 The Need for Custom Serialization

Standard Zenoh payloads are opaque byte arrays. For structured communication with CAN messages, we implemented a custom serialization layer:

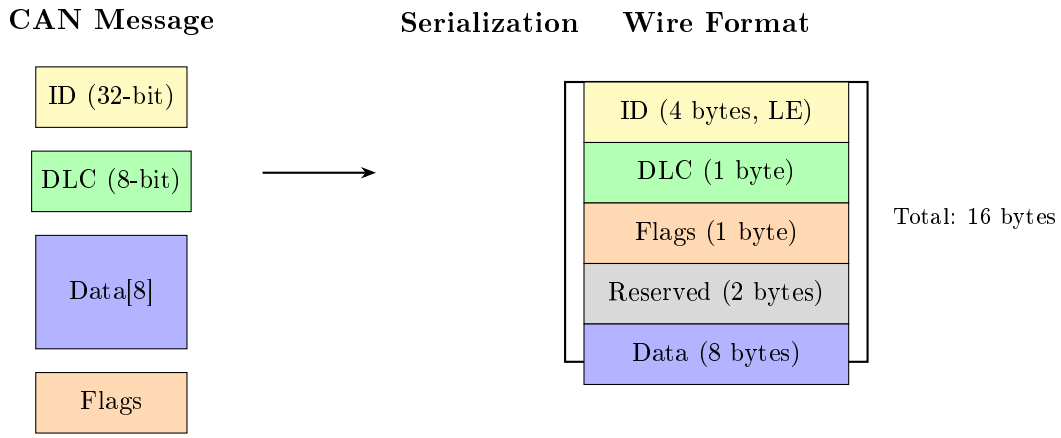

Figure 9: CAN message serialization format

5.3.2 Serialization Implementation

```
#ifndef ZENOH_SERIALIZE_H
#define ZENOH_SERIALIZE_H

#include <stdint.h>
#include <stddef.h>
#include "can_types.h"

#define ZENOH_CAN_MSG_SIZE 16

typedef struct __attribute__((packed)) {
```

```
    uint32_t id;        // CAN identifier
    uint8_t  dlc;       // Data length code
    uint8_t  flags;     // Extended ID, RTR, etc.
    uint16_t reserved;  // Alignment padding
    uint8_t  data[8];   // CAN data payload
} zenoh_can_frame_t;

/**
 * Serialize a CAN message for Zenoh transmission
 * @param msg Source CAN message
 * @param buf Destination buffer (min 16 bytes)
 * @param buf_len Buffer length
 * @return Number of bytes written, or -1 on error
 */
int zenoh_serialize_can_msg(
    const can_message_t *msg,
    uint8_t *buf,
    size_t buf_len
);

/**
 * Deserialize a Zenoh payload to CAN message
 * @param buf Source buffer
 * @param buf_len Buffer length
 * @param msg Destination CAN message
 * @return 0 on success, -1 on error
 */
int zenoh_deserialize_can_msg(
    const uint8_t *buf,
    size_t buf_len,
    can_message_t *msg
);

#endif // ZENOH_SERIALIZE_H
```

Listing 5: zenoh_serialize.h - Serialization interface

```
#include "zenoh_serialize.h"
#include <string.h>

int zenoh_serialize_can_msg(
    const can_message_t *msg,
    uint8_t *buf,
    size_t buf_len)
{
    if (!msg || !buf || buf_len < ZENOH_CAN_MSG_SIZE) {
        return -1;
    }

    zenoh_can_frame_t *frame = (zenoh_can_frame_t *)buf;

    // Use little-endian for cross-platform compatibility
    frame->id = msg->identifier;
    frame->dlc = msg->data_length_code;
    frame->flags = 0;

    if (msg->extd) {
        frame->flags |= 0x01;  // Extended ID flag
    }
    if (msg->rtr) {
        frame->flags |= 0x02;  // RTR flag
    }

    frame->reserved = 0;
```

```
    memcpy(frame->data, msg->data, 8);

    return ZENOH_CAN_MSG_SIZE;
}

int zenoh_deserialize_can_msg(
    const uint8_t *buf,
    size_t buf_len,
    can_message_t *msg)
{
    if (!buf || !msg || buf_len < ZENOH_CAN_MSG_SIZE) {
        return -1;
    }

    const zenoh_can_frame_t *frame =
        (const zenoh_can_frame_t *)buf;

    msg->identifier = frame->id;
    msg->data_length_code = frame->dlc;
    msg->extd = (frame->flags & 0x01) != 0;
    msg->rtr = (frame->flags & 0x02) != 0;
    memcpy(msg->data, frame->data, 8);

    return 0;
}
```

Listing 6: zenoh_serialize.c - Serialization implementation

5.4 Zenoh Node Core Implementation

```
#include "zenoh_node.h"
#include "zenoh_serialize.h"
#include <zenoh-pico.h>

static z_owned_session_t session;
static z_owned_publisher_t pub_can_rx;
static z_owned_subscriber_t sub_can_tx;

// Key expressions for CAN bridge
#define KE_CAN_RX "vehicle/can/rx"   // CAN -> Zenoh
#define KE_CAN_TX "vehicle/can/tx"   // Zenoh -> CAN

/**
 * Callback for messages received from Zenoh to send on CAN
 */
void can_tx_callback(const z_sample_t *sample, void *ctx) {
    can_message_t msg;

    if (zenoh_deserialize_can_msg(
            sample->payload.start,
            sample->payload.len,
            &msg) == 0) {
        // Forward to CAN bus
        can_transmit(&msg);
    }
}

/**
 * Initialize the Zenoh node
 */
esp_err_t zenoh_node_init(const zenoh_node_config_t *config) {
    // Configure Zenoh
```

```
    z_owned_config_t z_config = z_config_default();

    if (config->router_addr != NULL) {
        zp_config_insert(z_loan(z_config),
            Z_CONFIG_CONNECT_KEY,
            config->router_addr);
    }

    // Open session
    if (z_open(&session, z_move(z_config)) < 0) {
        return ESP_FAIL;
    }

    // Start read/lease tasks
    zp_start_read_task(z_loan(session), NULL);
    zp_start_lease_task(z_loan(session), NULL);

    // Declare publisher for CAN RX
    pub_can_rx = z_declare_publisher(
        z_loan(session),
        z_keyexpr(KE_CAN_RX),
        NULL
    );

    // Subscribe to CAN TX messages
    z_owned_closure_sample_t callback =
        z_closure(can_tx_callback, NULL, NULL);
    sub_can_tx = z_declare_subscriber(
        z_loan(session),
        z_keyexpr(KE_CAN_TX),
        z_move(callback),
        NULL
    );

    return ESP_OK;
}

/**
 * Publish a CAN message to Zenoh
 */
esp_err_t zenoh_publish_can_msg(const can_message_t *msg) {
    uint8_t buf[ZENOH_CAN_MSG_SIZE];

    if (zenoh_serialize_can_msg(msg, buf, sizeof(buf)) < 0) {
        return ESP_FAIL;
    }

    z_publisher_put_options_t options =
        z_publisher_put_options_default();

    if (z_publisher_put(
            z_loan(pub_can_rx),
            buf,
            ZENOH_CAN_MSG_SIZE,
            &options) < 0) {
        return ESP_FAIL;
    }

    return ESP_OK;
}
```

Listing 7: zenoh_node.c - Core node implementation (excerpt)

5.5 Data Flow Architecture

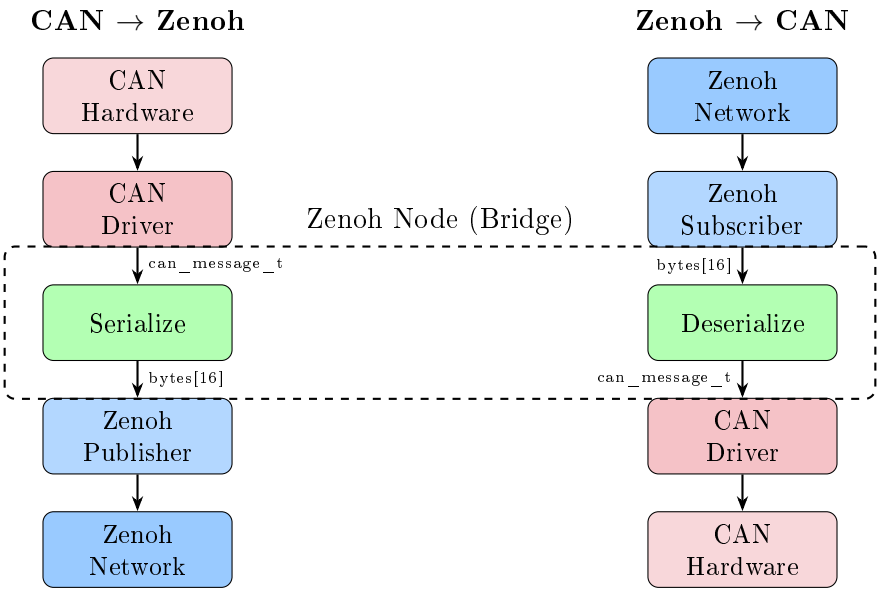

Figure 10: Bidirectional data flow through the Zenoh node

6 Configuration

6.1 Kconfig Options

```
menu "Zenoh Node Configuration"

    config ZENOH_NODE_ENABLED
        bool "Enable Zenoh Node"
        default y
        help
            Enable the Zenoh communication node.

    config ZENOH_ROUTER_ADDRESS
        string "Zenoh Router Address"
        default "tcp/192.168.1.1:7447"
        depends on ZENOH_NODE_ENABLED
        help
            Address of the Zenoh router to connect to.

    config ZENOH_NODE_MODE
        int "Zenoh Node Mode"
        default 0
        range 0 2
        depends on ZENOH_NODE_ENABLED
        help
            0 = Client, 1 = Peer, 2 = Router

    config ZENOH_PUBLISH_PERIOD_MS
        int "Publish period (ms)"
        default 100
        depends on ZENOH_NODE_ENABLED
        help
            Period for publishing CAN messages to Zenoh.

endmenu
```

Listing 8: Kconfig - ESP-IDF menuconfig options

7 Performance Considerations

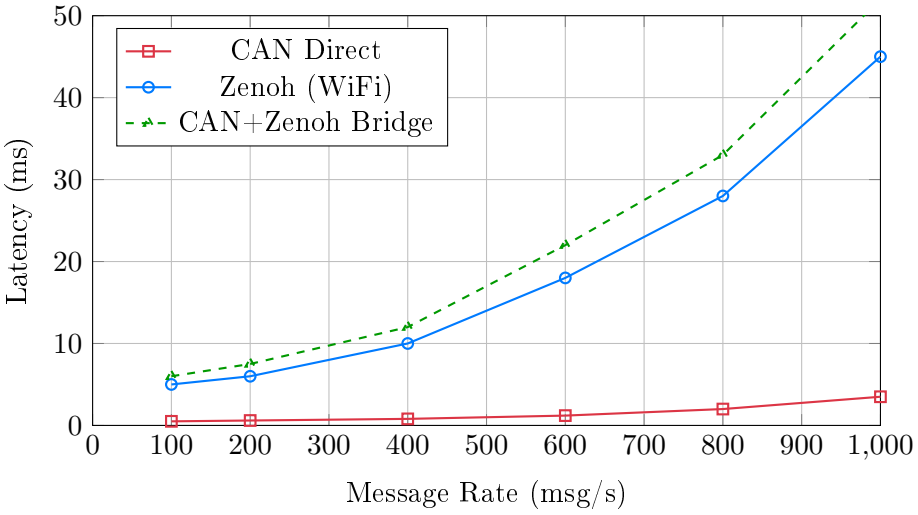

Figure 11: Latency comparison under different message rates (illustrative)

7.1 Optimization Strategies

1. **Message Batching**: Aggregate multiple CAN messages before Zenoh transmission
2. **Priority Filtering**: Only bridge high-priority CAN IDs
3. **Compression**: Enable Zenoh payload compression for high-bandwidth scenarios
4. **QoS Configuration**: Use best-effort for telemetry, reliable for commands

8 Conclusion

The Zenoh node implementation successfully bridges the gap between modern pub/sub communication systems and traditional CAN networks. Key achievements include:

- Successful integration of vendored zenoh-pico for ESP32
- Custom serialization protocol for efficient CAN message transport
- Bidirectional communication enabling high-level system integration
- Configurable operation via ESP-IDF Kconfig

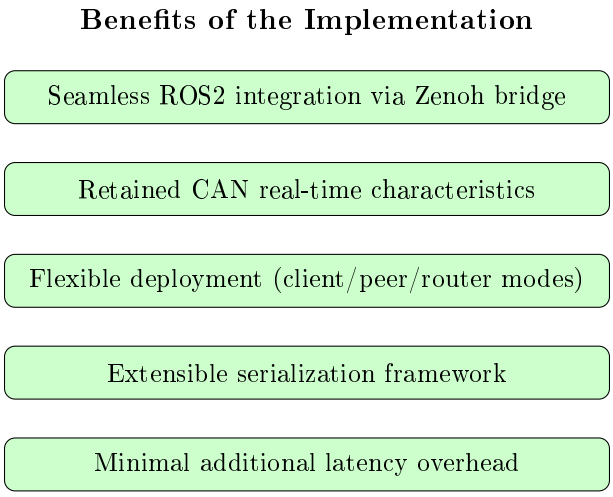

Figure 12: Summary of implementation benefits

A References

- Zenoh Documentation: `https://zenoh.io/docs/`
- zenoh-pico GitHub: `https://github.com/eclipse-zenoh/zenoh-pico`
- CAN 2.0 Specification: Bosch CAN Specification Version 2.0
- ESP-IDF Programming Guide: `https://docs.espressif.com/projects/esp-idf/`